FACULTAT D'INFORMÀTICA DE BARCELONA - FIB

UNIVERSITAT POLITÈCNICA DE CATALUNYA -UPC

Creación de un videojuego 3D educativo con metodología basada en la modificación de código

Jordi Orea Moya

Directora: Marisa Gil Gómez

Titulación: Ingeniería Informática

Creación de un videojuego 3D educativo con metodología basada en la modificación de código

Jordi Orea Moya

RESUMEN

Este proyecto se basa en la creación de un videojuego 3D orientado a la enseñanza y puesta en práctica de conocimientos de programación. Dichos conocimientos se proyectan en la metodología del juego, la cual se basa en modificar partes del propio código mediante una interfaz para que el personaje del juego pueda superar los diferentes niveles. Ésta metodología de modificación de código permite visualizar en tiempo real sus efectos gracias al ciclo de vida propio de un videojuego.

En este documento se detallan las partes de análisis, diseño e implementación del videojuego mediante el motor de desarrollo multiplataforma Unity. Se da un repaso general a todos los componentes que han sido necesarios en la creación del videojuego así como de los principales problemas y desafíos encontrados.

Por último, se expone una planificación temporal junto con las conclusiones extraídas de la realización del proyecto, además de dar indicaciones para posibles ampliaciones o modificaciones del videojuego resultante.

ÍNDICE

1. MOTIVACIÓN

Desde bien pequeño los videojuegos han sido una de mis grandes pasiones. Mis recuerdos de infancia siempre han estado asociados a las videoconsolas con las que tan buenos momentos pasé desde muy temprana edad. Ese fue uno de los principales motivos que me hicieron decantarme por estudiar la carrera de Ingeniería Informática.

Al escoger el proyecto final de mis estudios quise apostar por un trabajo que me permitiese adentrarme de una manera exhaustiva en la rama del sector por la que siento predilección.
Todo esto, junto con la idea expuesta por mi tutora, ha permitido que pueda elaborar un proyecto basado en mis pasiones incorporando a su vez una vertiente pedagógica que le aporta contenido y lo aleja de lo meramente lúdico.

La finalidad de este trabajo recae en poner en práctica mis conocimientos previos y adquirir nuevos a través de los retos que supone trabajar con una plataforma hasta el momento desconocida, como es Unity[1]. Además, mi aprendizaje servirá para que otras personas puedan disfrutar e iniciarse en el mundo de la programación. La incorporación de conocimientos relacionados con la informática unido a unas bases jugables me ha parecido un buen punto de partida para desarrollar un proyecto educativo interesante.

2. INTRODUCCIÓN

El contexto de creación del videojuego detallado en este proyecto se origina a raíz de pensar en algún tipo de mecánica jugable que involucrase, dentro del videojuego, la interacción con el propio código. Partiendo de esta premisa empecé a pensar ideas que le diesen herramientas al usuario, o jugador en este caso, para poder experimentar de alguna forma los cambios que el mismo efectuase dentro de los elementos del videojuego.

En base a una propuesta parecida de la tutora Marisa Gil, se ha desarrollado la idea de este videojuego que basa su jugabilidad en modificar partes del código de forma que se pongan en práctica los conocimientos que el usuario pueda tener sobre aspectos de la programación, a la vez que no pierde el espíritu lúdico propio de un videojuego. De esta forma, paralelamente, se permite al usuario ver de forma gráfica las modificaciones que realiza sobre el código, dándole una visión directa de los cambios provocados. Estos cambios se han diseñado para ser introducidos mediante una interfaz que permite modificar partes predefinidas del código que a su vez modifica el código real aprovechándose del ciclo de ejecución propio de un videojuego.

Respecto al diseño del videojuego, éste contiene un número definido de niveles, cuatro, en los que progresivamente se va aumentando la dificultad a la vez que se añade “temario” al código que se presenta. Estos son algunos ejemplos de interacciones que incorpora;

- Modificación, asignación y operaciones con variables que modifiquen el comportamiento del entorno.
- Uso de estructuras iterativas y métodos predefinidos.
- Estructuración de código.
- Concurrencia y paralelismo básico.

Solo se pueden modificar algunas partes del código bloqueando otras en función del puzle que se quiera plantear en esa fase. No es el objetivo, en ningún caso, crear un editor complejo que permita interpretar todo el lenguaje que se pueda introducir libremente.

Una de las cosas que se pretende es que el usuario experimente las consecuencias directas de la modificación del código de forma que pueda asimilar fácilmente y de forma entretenida los conocimientos que pone en práctica.

El público objetivo de esta aplicación serían jugadores con conocimientos básicos de programación que deseen probar un tipo diferente de videojuego donde los retos no vengan dados por agilidad visual o precisión, como muchos de los videojuegos tradicionales, si no por retar a poner a prueba sus nociones básicas de programación y entendimiento de código.

3. ESTADO DEL ARTE[18]

El auge de los videojuegos ha hecho que, en un periodo de tiempo relativamente corto, éstos den el paso hacia nuevas formas de narrativa y mecánicas jugables. Se ha pasado de los simples juegos de plataformas y disparos con los que nació el mundo de los videojuegos, hace más de 30 años, a innovadoras formas de jugabilidad centradas en, por ejemplo, contar una historia o hacer llegar al jugador experiencias basadas en la sensibilidad y emoción de las imágenes generadas.

Una de las últimas vertientes viene asociada al concepto de "gamificación"[2][3], el cual hace referencia a usar metodologías o características propias del mundo lúdico en otros aspectos de nuestra vida. Normalmente se asocia con la educación, que es la parte que nos interesa para contextualizar este proyecto. Éste proyecto no está enfocado tanto a la vertiente de enseñanza y educación como podrían ser los programas educativos para niños y jóvenes como Scratch[4].

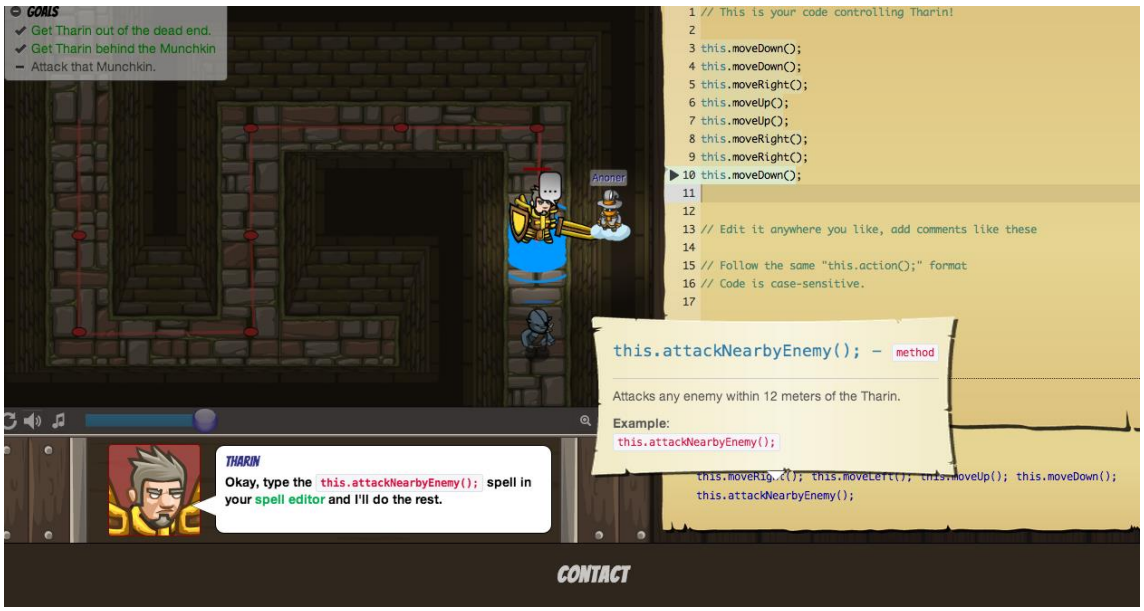
Éste es el representante más significativo de una corriente de programas (no son del todo videojuegos) enfocados en el aprendizaje de la programación u otros conocimientos similares, normalmente de carácter infantil.

Por otra banda, hay unos pocos juegos que basan su jugabilidad en integrar principios de programación o informática que pongan a prueba los conocimientos ya adquiridos del jugador, dejando más de lado la vertiente puramente formativa. Este tipo de juegos son una auténtica minoría. Para la concepción de este proyecto se han tenido en cuenta dos de los que he encontrado más elaborados.

3.1. CODECOMBAT[5]

Una de las más elaboradas plataformas para la enseñanza de programación unida a unas mecánicas propias de los videojuegos clásicos de mazmorras. Se basa en mover a nuestros personajes por unos tableros con casillas hasta que encuentren la salida y acaben con los enemigos correspondientes. Absolutamente todos los movimientos se hacen modificando un editor de código que ocupa la parte derecha de la pantalla de juego.

Está orientado en el aprendizaje de Python y JavaScript. Empieza desde los principios más básicos de la programación hasta llegar a niveles muy profundos debido a su complejidad y profundidad. Está pensado para durar muchas horas.



Aunque desconocía de su existencia antes de proponer este proyecto, CodeCombat comparte algunas cosas en común con mi proyecto. La finalidad de cada nivel es ligeramente parecida (acabar con todos los enemigos y llegar a la salida) así como la disposición fija de un editor de código como principal entrada de las órdenes del usuario. Su mayor diferenciación proviene de

su énfasis en el aprendizaje y la complejidad del editor, que implementa un compilador de lenguaje real.

3.2. HACK'N'SLASH[6]

Este videojuego de PC de la productora Double Fine quizá es el más conocido de los aquí expuestos. Su jugabilidad se basa en mover a un personaje por escenarios amplios y eliminar a los enemigos "hackeando" el código que los genera. De esta forma se puede modificar la vida de los enemigos a cero, hacer que los ataques hagan menos daño al jugador o proporcionar más distancia de salto.

Aunque no he tenido la oportunidad de probarlo, por lo que he podido ver mediante imágenes, éste juego está cerca de mi idea respecto a aplicar conocimientos sobre un ámbito (programación en este caso) a unas mecánicas jugables que predominan respecto al aprendizaje. Parece que ambos juegos comparten similitudes respecto a mecánica y situaciones planteadas.

3.3. CODEHUNT[7]

Este último juego investigado proviene de Microsoft y está enfocado en resolver pequeños trozos de código que esperan una determinada salida. En su concepción no está muy alejado de éste proyecto pero lo distingue principalmente su interfaz basada puramente en editores de código, eliminando cualquier atisbo de los otros elementos que suelen identificar a un videojuego.

Como puede verse, este proyecto presenta aspectos jugables existentes en unos pocos exponentes de estos casos de juegos que referencian al mundo de la programación y pretende darle un ligero componente educativo, intentando aunar el mundo de los videojuegos y la educación.

4. DESCRIPCIÓN DE LOS OBJETIVOS

Sabiendo el contexto en el que se ha ubicado la filosofía que hay detrás del videojuego se puede desgranar el objetivo del proyecto, crear un videojuego de carácter educativo relacionado con la informática, en otros objetivos más concretos.

- Diseñar y desarrollar un videojuego en 3D que base su jugabilidad en permitir al usuario modificar partes del código del propio juego para poder avanzar.

- Demostrar de forma visual e inmediata los efectos de cambios en el código de un programa.

- Desplegar el videojuego como aplicación web para que pueda ser más fácilmente accesible.

- Como objetivo secundario podríamos incluir el aprendizaje propio de un nuevo motor, Unity, para el cual tenía muchas ganas de desarrollar.

5. METODOLOGÍA DE TRABAJO

5.1. HERRAMIENTAS EMPLEADAS

Unity 5.3: Motor de creación de videojuegos y principal herramienta para el desarrollo del proyecto. Posee una versión de licencia gratuita (siempre que no se sobrepase un elevado margen de beneficios con el producto desarrollado) y con posibilidad de crear juegos para multitud de plataformas. Permite trabajar con 3 lenguajes de programación (C#, JavaScript y Boo) de los cuales se ha escogido JavaScript para el proyecto. No se ha optado por la última versión del programa ya que ocasiona problemas con el despliegue automático en la nube.

MonoDevelop: Editor de código oficial de Unity. Es sencillo y permite integración directa con Unity. Solo hay que guardar los cambios en el editor para que Unity haga la compilación de los scripts afectados automáticamente. Se ha preferido en contra de editores más potentes (Visual Studio, Visual Code, etc) por su solvencia y sencillez de integración.

Notepad++: Editor de texto usado de forma auxiliar para ediciones de texto y código que no tuviesen que ser necesariamente integradas en Unity inmediatamente.

Unity Cloud Building: Servicio gratuito de Unity para compilar y generar los binarios de forma automática y almacenarlos en la nube. Solo precisa de algún tipo de repositorio online donde obtener el código.

Git: Software de control de versiones que permite tener repositorios locales y remotos. Se ha optado por él porque estaba familiarizado con su uso, además de por su potencia y versatilidad.

BitBucket: Servicio de alojamiento web de repositorio de código que permite trabajar con proyectos que usen Mercurial o Git como control de versión. Su uso es necesario para tener una copia de seguridad fuera de la máquina local así como para poder tener integración continua mediante Unity Cloud Building.

SourceTree: Cliente gratuito para la gestión de repositorios Git y Mercurial. Se ha usado para tener una interfaz gráfica que facilite los guardados (commits) de código y el envío al repositorio remoto de BitBucket.

MagicaVoxel[8]: Editor ligero gratuito basado en creación de modelos mediante voxels de 8 bits. Ha sido una herramienta perfecta para crear modelos “pixelados” de forma rápida y sencilla.

Trello: Herramienta web de gestión de proyectos, con versión gratuita y de forma muy directa. Su facilidad permite implementar una metodología ágil, como puede ser Scrum, de forma más o menos acertada. Su principal baza es ofrecer una gestión y planificación de tareas simple, en contraposición a otras herramientas que requieren más dedicación y recursos, como Jira.

Google Drive: Herramientas de edición y almacenaje en la nube, usadas principalmente por su comodidad y portabilidad para mantener archivos y apuntes de documentación juntos.

Cacoo: Editor de diagramas y flujos online que permite almacenar las creaciones y compartirlas fácilmente. Se ha usado brevemente para crear los diagramas de este documento ya que estaba familiarizado con él.

5.2. DESARROLLANDO CON UNITY

Unity es un potente motor de desarrollo de videojuegos creado por Unity Technologies. Es una herramienta disponible en los tres principales sistemas operativos (Microsoft Windows, OS X y Linux) y cuenta con diferentes versiones disponibles. Una de ellas gratuita, Unity Personal, y el resto de pago; Unity Plus, Unity Pro y Unity Enterprise, cada una de ellas con funcionalidades extra. En el momento de la redacción de este documento, la última versión estable es la 5.4.

Su versatilidad permite generar el videojuego creado para múltiples plataformas. Esto unido a su rendimiento, facilidad de uso y gran comunidad de usuarios, le han llevado a ser uno de las principales frameworks de creación de videojuegos independientes e incluso grandes empresas han empezado a utilizarlo de forma recurrente para sus producciones. Algunos ejemplos de ello son Capcom, Square Enix, Ubisoft,etc.[19]

Mi creciente interés por esta plataforma y su proliferación en el mercado laboral actual, junto con su accesibilidad, me hicieron decantarme por ella casi sin consultar las otras alternativas. Su principal competidor directo, Unreal Engine, en su última versión (4.0) ha sido abierto al público masivo ofreciéndose como aplicación gratuita. A pesar de ello, su vertiente más enfocada al resultado profesional y su mayor dificultad para conseguir material de aprendizaje lo han descartado.

Otras alternativas podrían ser herramientas enfocados al desarrollo de videojuegos en el ámbito de las dos dimensiones como Cocos2D, las cuales han sido descartadas por intentar elaborar un videojuego más complejo.

Finalmente se ha optado por la versión 5.3.6f1 de Unity debido a algunos inconvenientes encontrados a lo largo del proyecto que han hecho cambiar de versión de Unity hasta en dos ocasiones. Más adelante se hará referencia a los motivos que propiciaron estos cambios.

5.3. METODOLOGÍA EMPLEADA

Debido a las características propias del desarrollo de un videojuego, el cual es mucho más iterativo y dinámico que un desarrollo de un aplicativo clásico, se ha optado descaradamente por emplear una metodología ágil. En concreto se ha usado una adaptación libre de Scrum[9]. Ésta popular metodología está basada en sprints de una duración concreta y corta (generalmente una o dos semanas) que intentan delimitar los objetivos más cercanos a conseguir. Para ello se desglosan los objetivos principales en tareas denominadas "épicas" que, a su vez, se desglosan en tareas más atómicas y con mayor facilidad para estimarlas temporalmente.

Debido a que la creación del videojuego ha sido llevado a cabo por un solo desarrollador, un servidor, se han eliminado elementos teóricos relativos a la metodología Scrum como pueden ser el Product Owner o el Scrum Master. Esto se ha hecho en beneficio de un desarrollo más rápido y coherente con el desarrollo de una sola persona. Cuando ha habido un contratiempo, este ha podido ser tratado con mayor inmediatez que si se hubiese usado una metodología clásica. Las pruebas han ido haciéndose a medida que se iban desarrollando partes y elementos del juego ya que el propio ciclo de un videojuego induce a ello. Por ejemplo, al implementar una característica de la interfaz o un elemento posicionado en el escenario, seguidamente se ha pasado a comprobar su validez ejecutando el propio juego y verificando que cumple con los requisitos esperados. A su vez, también se van generando pequeñas pruebas de integración dependiendo del alcance de los cambios.

Toda la gestión del código se ha efectuado mediante un repositorio Git almacenado en BitBucket. De esta forma había doble seguridad de integridad de los datos en caso de querer recuperar una versión anterior o (afortunadamente no ha pasado) pérdida de datos.

5.4. PLANIFICACIÓN INICIAL

Teniendo esta metodología en mente se desarrolló una planificación inicial al poco de empezar el proyecto que, como se podrá comprobar al final del documento, ha sufrido desviaciones importantes en algunos apartados.

FASE 1 (ABRIL / MAYO)

La primera fase del proyecto está planificada para el aprendizaje del motor Unity, ya que mis conocimientos prácticos con el eran prácticamente nulos. También incluían las pruebas conceptuales para ser que sería capaz de conseguir con esta herramienta.

Planificación y preparación		
	Creación de propuesta	
	Listar tareas	
	Planificar tareas	
	Redactar propuesta final	
	Redactar informe	
Preparación - Aprendizaje		
	Aprendizaje motor Unity	
		Aprendizaje básicos y 2D
		Aprendizaje 3D
		Aprendizaje avanzado
		Aprendizaje específico a los imprevistos del proyecto
	Profundización en Javascript	
	Investigar posibidad de interacción con web	
	Investigar uso de threads en Unity	

FASE 2 (JUNIO / JULIO)

La segunda fase corresponde al diseño e implementación de los principales componentes del videojuego así como del primer nivel del videojuego.

Creación de repositorio		
	Configuración de repositorio	
	Subida proyecto inicial	
Pruebas de despliegue (build)		
	Despliegue convencial	
	Despliegue en web	
Editor de código		
	Análisis de requisitos y componentes	
	Diseño de funcionalidades	
	Implementación básica como prototipo	
	Pruebas y limitaciones	
	Revisar requisitos y diseño	
Prueba de concepto inicial		
	Escena inicial	
	Integración del editor	
	Analisis de limitaciones y posibilidades	
Diseño del juego		
	Concretar mecánica y objetivos	
	Diseñar fases con ejemplos	
		Fase inicial
		Fases avanzadas (2..N)
	Diseñar uso del editor en el juego	
	Incorporar soluciónes a las fases	

FASE 3 (JULIO / AGOSTO)

La tercera fase engloba la creación de toda la parte gráfica y desarrollo del resto de componentes del juego. Pasar de los prototipos iniciales a incluir la carga gráfica que sería casi final.

Assets		
	Modelos de escenario	
	Modelo de personajes	
	Modelos de enemigos	
	Otros assets	
	Conseguir efectos de sonido	
	Conseguir música	
Implementación		
	Escenarios	
	Personajes principales	
	Enemigos	
	IA	
	Otros / bonificadores	
	UI	
	Implementar editor definitivo	
	Pruebas de integración	

FASE 4 (AGOSTO)

La última fase se reserva para la finalización del resto de niveles, la incorporación de audio y el despliegue web.

	Implementar resto de fases diseñadas (iterativo)	
	Efectos de sonido	
	Música	
Implementación parte web		
	Definición de entorno	
	Mostrar código de ejemplo en web	
	Integrar juego con entorno	
	Pruebas y revisión	
Pruebas generales de integración		
Pruebas finales con usuarios		

6. DISEÑO DEL VIDEOJUEGO

6.1. REQUISITOS

Debido a que no había unos requisitos específicos que vengan definidos y delimiten la versión creativa del juego, como si hay al crear una versión de un juego físico o digital existente o al realizar el diseño de un aplicativo encargado por una persona ajena, se han establecido algunos requisitos funcionales o directrices que delimitan el alcance del proyecto.

RESTRICCIÓN

Durante la fase de aprendizaje con el desarrollo de los tutoriales de Unity ya se gestó esta restricción de diseño:

- El videojuego debe ser creado con modelados y metodologías propias de un juego 3D.

Simplemente creía que el desarrollo de un juego en 3D, además de ser más estimulante, podía ofrecer una riqueza jugable que le diese mayor entidad que uno realizado con una perspectiva y mecánica en 2D.

REQUISITOS FUNCIONALES

Estos son los principales requisitos funcionales que se han especificado a la hora de empezar a diseñar el videojuego.

- Debe incorporar un editor de código que sirva como entrada de los datos del usuario
- Mantener una estructura en fases que permitan sensación de avance
- Presentar situaciones que se deban resolver mediante modificación de código
- Evitar mecánicas que distraigan del objetivo principal

REQUISITOS NO FUNCIONALES

Paralelamente a los requisitos funcionales, también han surgido una serie de requisitos que definen la calidad y características que se quieren conseguir para poder asimilar con éxito los requisitos funcionales.

- El videojuego será generado como un aplicativo web.
- La metodología de juego debe ser accesible y fácil de asimilar, en consonancia con el propósito del videojuego.
- El aspecto visual debe ayudar a esa accesibilidad comentada, aunque en ningún momento debe ser algo primordial.
- El videojuego debe ser eficiente en sus recursos para poder ejecutarse en cualquier tipo de máquina con unas características ordinarias.

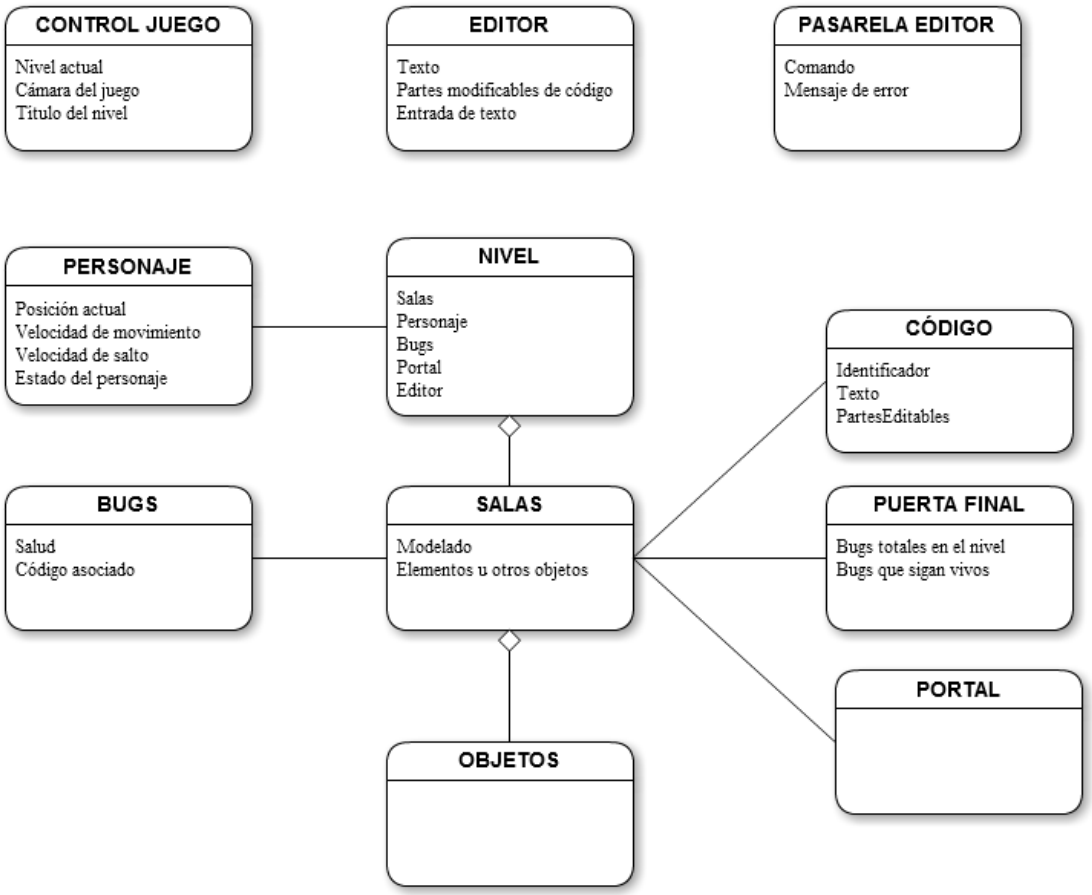
Con todo ello, a continuación se muestra el resultado del diseño elaborado para este videojuego.

6.2. ARGUMENTO

La ambientación del videojuego se sitúa en las entrañas de un programa representado como estancias virtuales en forma de salas y pasillos. El personaje tiene un rol de vigilante o agente de seguridad y por ello debe buscar y eliminar a los enemigos, los cuales representan “bugs” que se han colado dentro del código del programa. A través de los diferentes niveles (que representan distintos programas) irá limpiando los códigos hasta que no quede ningún bug suelto! El nombre escogido para el videojuego es “Bug Finder” en relación a la temática explicada.

6.3. MAPA CONCEPTUAL DE LOS DATOS

De forma libre, y poco ortodoxa, se ha usado el siguiente modelo conceptual para representar gráficamente los elementos que son necesarios para poder elaborar el juego que satisfaga los requisitos funcionales anteriormente descritos. En el desarrollo del juego han ido saliendo otros elementos, fruto de la necesidad o de la creatividad en otros casos, pero este modelado sirve como base para situarse y entender las mecánicas en que se basa el juego.



A continuación, se listan los elementos del modelo junto con una breve descripción de su funcionalidad.

Nota: Aunque no deberían aparecer en este apartado del documento debido a que no pertenecen a la misma etapa de desarrollo del proyecto, se ha optado por integrar imágenes finales de los diferentes elementos descritos con el objetivo de que le sea más fácil al lector relacionar los términos descritos. Así, en capítulos posteriores, tendrá una visión más clara de los elementos jugables que se describen.

PERSONAJE

El personaje es el elemento principal del videojuego, el cual es controlado por el jugador. El objetivo de cada nivel consiste en llevar al personaje a la salida intentando no caer por ningún precipicio.

ATRIBUTOS

Posición actual

Velocidad de movimiento

Velocidad de salto

Estado del personaje: saltando o en tierra

NIVEL

El nivel es donde transcurre la acción del videojuego. En él se encuentran todos los elementos que han sido diseñados para esa parte del videojuego.

ATRIBUTOS

Salas que los componen

Personaje

Bugs

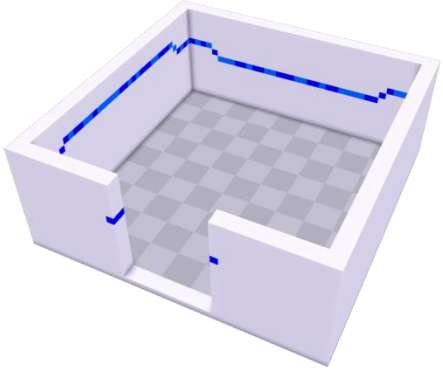
Portal

Editor

Puertas y otros elementos específicos de cada nivel

SALA

Una sala o estancia hace referencia a cada uno de los elementos que constituyen en conjunto el escenario de un nivel. Dentro de ellas se sitúan los elementos definidos para ese nivel con los que el personaje puede interactuar.



ATRIBUTOS

Modelado

Elementos u otros objetos

TROZOS DE CÓDIGO

Los trozos de código son objetos coleccionables que otorgan al jugador las partes del texto del código que se pueden modificar con el editor.

ATRIBUTOS

Identificador

Texto

PartesEditables

EDITOR

El editor es una interfaz que permite establecer la entrada de los comandos del jugador y modificar los elementos del nivel en consonancia con los parámetros introducidos.

ATRIBUTOS

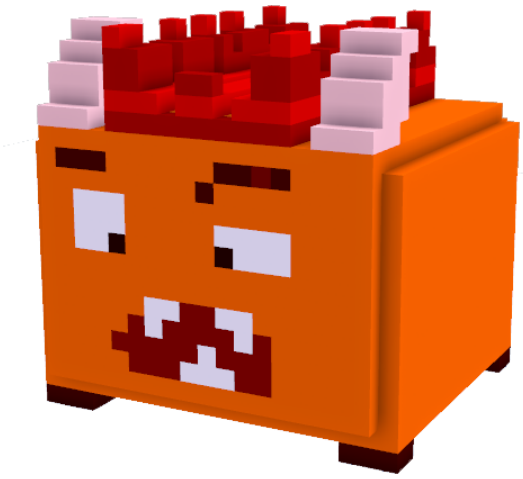
Texto cargado hasta el momento en ese nivel

Partes modificables de código conseguidas

Entrada de texto para el jugador

BUGS

Los bugs o enemigos representan el objetivo a eliminar en cada uno de los niveles. La forma de destruirlos proviene de la modificación de sus parámetros o partes de código que tengan asociados. Si no se eliminan todos, no se puede pasar al siguiente nivel.



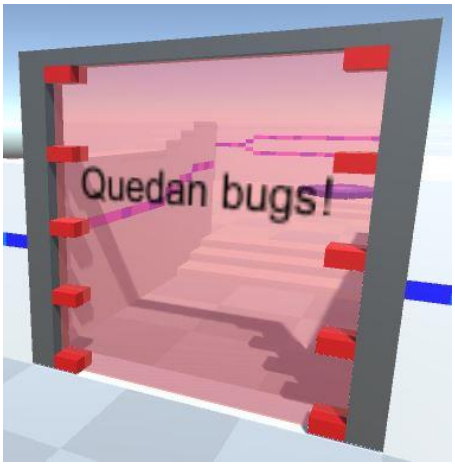
ATRIBUTOS

Salud

Código asociado para poder eliminarlo

PUERTA FINAL

La puerta final está situada en la estancia de cada escenario que otorga la salida hacia el siguiente nivel. Su funcionalidad es comprobar que se hayan eliminado todos los bugs de dicho nivel antes de que el personaje pueda acceder al siguiente.



ATRIBUTOS

Número de bugs totales en el nivel

Número de bugs que sigan vivos

PORTAL

El portal es el elemento que permite trasladar al personaje al siguiente nivel del juego. Su posición siempre viene definida por la última sala del escenario e incluye una puerta final previa para asegurar que se han cumplido los objetivos.

ATRIBUTOS

-

PUERTAS Y OTROS ELEMENTOS

Las puertas, plataformas y otros elementos del nivel que varían de uno a otro según el desafío o situación que se quiera plantear al jugador. Cada uno de ellos tiene unas características propias pero comparten, en su mayoría, la posibilidad de ser modificados.

ATRIBUTOS

Posición, velocidad, etc, dependiendo del tipo de elemento.

CONTROL JUEGO

Clase lógica que no representa a ningún elemento físico dentro del videojuego. Contiene toda la carga de datos e información necesaria para ir actualizando la dinámica del juego según los movimientos y acciones del jugador.

ATRIBUTOS

Nivel actual
Cámara del juego
Título del nivel
Referencias a multitud de parámetros de otros elementos

PASARELA EDITOR

Clase auxiliar que sirve como enlace entre la clase de Editor y los elementos a los que se hace referencia con las modificaciones de código. Su función es pasar los datos entre clases y validar que la información que se introduce es correcta.

ATRIBUTOS

Comando introducido
Mensaje de error en caso de que no sea válido el comando pasado

6.4. METODOLOGÍA JUGABLE

El objetivo en el juego es destruir todos los “enemigos” (que representan bugs de ese mundo virtual) para poder acceder al siguiente nivel. El desarrollo de un nivel del videojuego consiste en mover a nuestro personaje por un entorno con diferentes estancias en busca de objetos, que representan partes de código, sorteando obstáculos y trampas.

Cuando encuentra un objeto, se añade una parte de código a la interfaz que simula una consola. En dicha interfaz, se va mostrando el código obtenido para ese nivel y el jugador puede modificar las partes de código que se le permita, para poder acabar con los enemigos existentes y alcanzar la sala que contiene un portal con la salida del nivel.

Solo se pueden modificar algunas partes del código que sean previamente conseguidas por el jugador, bloqueando otras en función de la situación que se quiera plantear al jugador en ese nivel.

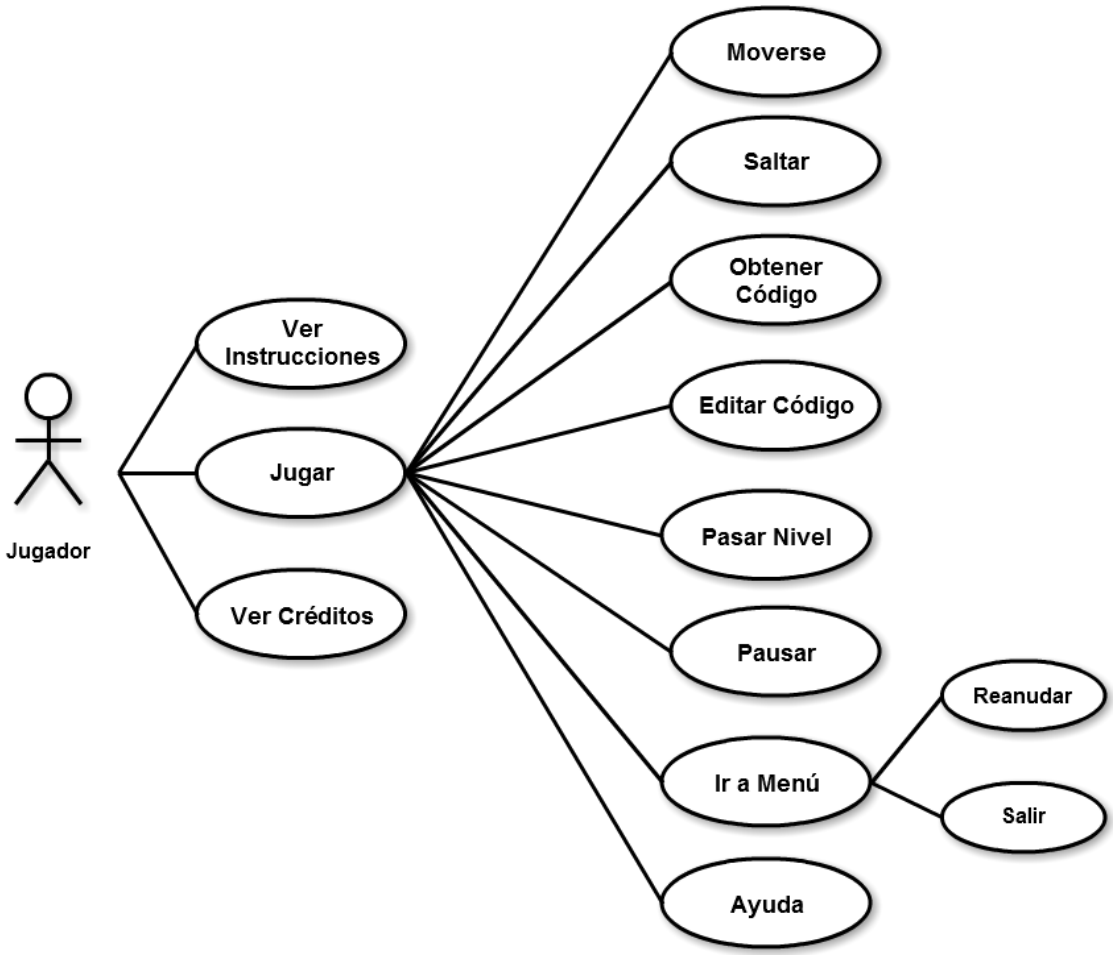
Al jugador se le proponen situaciones diseñadas para que las deba solucionar modificando el código del juego mediante los pequeños trozos de código que va obteniendo distribuidos por el escenario. Cuando modifica un comando del código, se verifica que tenga sentido y esté dentro de unos parámetros válidos, y se modifica el objeto del juego al que hace referencia. Éste puede ser una puerta que impida el paso, una plataforma que permita avanzar o los propios enemigos que pueden ser eliminados disminuyendo (vía modificación de la variable en el código) su propia vida a cero.

Las situaciones son distintas en cada nivel dotándolo de una sensación de progreso y aventura. Como se ha comentado anteriormente, se aleja de videojuegos de puzles clásicos donde en cada nivel se hace exactamente la misma mecánica (como por ejemplo Tetris).

El personaje tiene un poder que, mediante un botón o elemento de la interfaz, permite parar la acción para que el jugador pueda examinar el código y modificarlo con calma, sin tener que estar pendiente del estado actual de los enemigos o trampas.

6.5 MODELO DE CASOS DE USO

En base a las mecánicas anteriormente citadas se ha definido un diagrama que englobe todas estas acciones como casos de uso. Al tratarse de un videojuego es una forma atípica de usar un diagrama de este tipo, pero creo que puede resultar útil para dar una visión general del alcance de las posibilidades implementadas en el videojuego. Obviamente la interacción entre ellas es lo que ocasiona la riqueza jugable y le da sentido a la experiencia del videojuego, lo cual es difícil de captar en una documentación convencional.



6.6 LISTADO DE CASOS DE USO

De forma resumida se detallan los diferentes casos de uso que componen el modelo presentado anteriormente. A su vez sirven como referencia de los estados en los que se puede encontrar una partida del juego.

CASO DE USO: VER INSTRUCCIONES

DESCRIPCIÓN

El jugador puede ver las instrucciones antes de empezar a jugar para así conocer las mecánicas generales en que se basa el juego. El acceso se realiza mediante la interfaz del menú principal del juego. Su uso es especialmente recomendado si es la primera vez que se dispone a jugar.

ACTORES

Jugador

PRECONDICIONES

Estar en el menú principal del juego.

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador selecciona la opción de “Instrucciones” desde la pantalla principal del juego.

{Mostrar instrucciones}

2. El sistema muestra una pantalla con un texto explicativo de las mecánicas y objetivos que ha de cumplir el jugador en cada nivel.

FLUJO ALTERNATIVO: SALIR DE LA PANTALLA

En **{Mostrar instrucciones}** el jugador selecciona el botón de “Salir”.

1. El sistema cierra la pantalla y vuelve al menú principal del juego.

CASO DE USO: JUGAR

DESCRIPCIÓN

El jugador comienza un nivel, bien sea desde el menú principal del juego o porque ha finalizado el nivel en el que se encontraba. Dentro de él podrá hacer los casos de uso que se listan a continuación. Representan la mayor parte de las principales mecánicas diseñadas.

ACTORES

Jugador

PRECONDICIONES: -

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador selecciona uno de los niveles desde el menú principal.

{Jugar}

2. El sistema carga el nivel seleccionado con todos los elementos del escenario y los archivos de código editables correspondientes.
3. El jugador puede hacer uso de cualquiera de los siguientes casos de uso.

FLUJO ALTERNATIVO: -

CASO DE USO: VER CRÉDITOS

DESCRIPCIÓN

El jugador puede ver los créditos de las personas y entidades involucradas en el proyecto.

ACTORES

Jugador

PRECONDICIONES

Estar en el menú principal del juego.

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador selecciona la opción “Créditos” desde la pantalla principal del juego.

{Mostrar créditos}

2. El sistema mueve la cámara para mostrar la pantalla donde se visualizan los créditos.
3. El jugador selecciona la opción de “Volver” para acceder de nuevo a la pantalla principal.

FLUJO ALTERNATIVO: -

CASO DE USO: MOVERSE

DESCRIPCIÓN

El jugador puede mover al personaje del juego mediante los inputs declarados en la configuración. Por defecto, es la clásica configuración de teclas WASD. Las limitaciones al movimiento vienen dadas por los muros y objetos del escenario que provocan colisiones impidiendo el paso.

ACTORES

Jugador

PRECONDICIONES: -

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador acciona alguno de los inputs asignados para el movimiento del personaje.

{Movimiento del personaje}

2. El sistema mueve al personaje respecto a su posición en el frame anterior.
3. Vuelve al punto 1.

FLUJO ALTERNATIVO: COLISIÓN CON UN OBJETO

En **{Movimiento del personaje}** se colisiona con uno de los elementos del escenario. Exceptuando los objetos que representa código ya que se obtienen al tocarlos, sin que ello ocasione un parón en el movimiento.

1. El sistema no actualiza la posición del personaje.
2. Vuelve al punto 1 del flujo principal.

FLUJO ALTERNATIVO: CAÍDA DEL ESCENARIO

En **{Movimiento del personaje}**, dicho movimiento provoca que el personaje salga de los límites del escenario.

1. El sistema simula las físicas de gravedad provocando la caída del personaje hasta que sobrepasa un límite de altura inferior respecto al escenario y se reinicia el nivel.

CASO DE USO: SALTAR

DESCRIPCIÓN

El jugador puede hacer saltar al personaje mediante el botón asignado, que es el “Espacio”. Una vez que se ha saltado, se puede seguir moviendo pero el personaje se ve afectado por una simulada gravedad.

ACTORES

Jugador

PRECONDICIONES

Que no se esté ejecutando actualmente el caso de uso **{Saltar}**

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador acciona el botón asignado para el salto del personaje.

{Salto del personaje}

2. El sistema aplica unas fuerzas al cuerpo del personaje que hacen simular el efecto de salto.
3. Al aterrizar, el personaje vuelve a su comportamiento habitual.

FLUJO ALTERNATIVO: -

CASO DE USO: OBTENER CÓDIGO

DESCRIPCIÓN

Mediante el movimiento del jugador, el personaje colisiona con un objeto marcado como "Coleccionable" e incorpora el código que tiene asociado al texto actual del editor del juego.

ACTORES

Jugador

PRECONDICIONES: -

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador hace colisionar al personaje con un objeto de código del escenario.

{Obtener Código}

2. El sistema copia el texto asociado a ese objeto de código, en el editor del juego. Simultáneamente crea y resalta las partes que pueden ser modificadas por el jugador.

FLUJO ALTERNATIVO: -

CASO DE USO: EDITAR CÓDIGO

DESCRIPCIÓN

El jugador puede seleccionar una de las partes del código visibles en el editor porque ya han sido previamente incorporadas al conseguir un objeto de código mediante el caso de uso {**Obtener código}**. Una vez seleccionado puede modificarlo mediante un área de entrada de texto y enviarlo al sistema para que lo modifique y el entorno del nivel cambie en consonancia a la modificación realizada.

ACTORES

Jugador

PRECONDICIONES

Que se haya ejecutado, como mínimo una vez, el caso de uso **{Obtener código},** es decir, que ya se haya recolectado al menos una parte de código que sea modificable por el jugador.

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador selecciona una de las partes de código que están marcadas en el editor como seleccionable.

{Editar código}

2. El sistema pausa el juego para que se pueda modificar tranquilamente.
3. El jugador modifica en un campo del editor, el texto correspondiente al trozo seleccionado.

{Validar código}

4. El sistema recoge el texto introducido en la interfaz del editor y lo verifica para ver si el comando introducido es válido.
5. Si el comando es válido, se realiza la modificación en la clase u objetos correspondientes y se vuelve a reanudar la acción del juego.

FLUJO ALTERNATIVO: EL COMANDO INTRODUCIDO NO ES VÁLIDO

Si en el punto 4 de **{Validar código}** el sistema valida el comando y recibe como respuesta de la validación que no es válido, se notifica al jugador.

1. El sistema muestra por pantalla un mensaje de error notificando que el comando introducido no es válido con una pequeña indicación o sugerencia para corregirlo.
2. El sistema no modifica ninguna parte del juego afectada.
3. Vuelve al punto 2 de **{Editar código}**.

FLUJO ALTERNATIVO: EL JUGADOR REANUDA LA ACCIÓN

Si en el punto 3 de **{Editar código}**, el jugador selecciona el botón de pausa.

1. El sistema quita la selección de código actual y reanuda la acción del juego.

CASO DE USO: PASAR DE NIVEL

DESCRIPCIÓN

Al llegar al final de un nivel, mediante el acceso al objeto “Portal” con el personaje, el sistema carga el siguiente nivel y actualiza los datos necesarios para jugar dicho nivel desde cero.

ACTORES

Jugador

PRECONDICIONES: -

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador llega con el personaje a la sala final del nivel y entra dentro del objeto “Portal”.

{Pasar de nivel}

2. El sistema carga el siguiente nivel y lo inicializa correctamente.

FLUJO ALTERNATIVO: FASE FINAL

Si en el punto 2 de **{Pasar de nivel}** nos encontramos con que hemos llegado al final del juego, volvemos al menú principal.

1. El sistema muestra el texto de celebración de victoria.

CASO DE USO: PAUSAR EL JUEGO

DESCRIPCIÓN

El sistema para la acción del juego para que el jugador pueda realizar una pausa.

ACTORES

Jugador

PRECONDICIONES: -

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador selecciona el botón de pausa o selecciona una parte modificable del editor de código.

{Pausar el juego}

2. El sistema pausa la acción del juego eliminando mediante la eliminación del tiempo transcurrido en la lógica del juego entre frames.

FLUJO ALTERNATIVO: EL SISTEMA ESTÁ PAUSADO

Si ya se había ejecutado este caso de uso y el estado actual del juego es pausado:

1. El sistema reanuda la lógica del juego.

CASO DE USO: IR AL MENÚ DE PAUSA

DESCRIPCIÓN

El jugador ejecuta mediante el botón predeterminado de “Escape”, la transición al menú del nivel donde, además de pausar el juego, se da la posibilidad de salir y volver al menú principal.

ACTORES

Jugador

PRECONDICIONES: -

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador ejecuta el botón configurado para ir al menú genérico de cada nivel.

{Ir al Menú de pausa}

2. El sistema pausa el juego ejecutando el caso de uso **{Pausar el juego}**.
3. El sistema muestra las diferentes acciones que puede realizar el jugador en este menú.

FLUJO ALTERNATIVO: -

CASO DE USO: REANUDAR EL JUEGO

DESCRIPCIÓN

El jugador reanuda la acción una vez que ya estaba el juego pausado desde el menú de pausa.

ACTORES

Jugador

PRECONDICIONES

Estar actualmente en el menú de pausa.

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador selecciona la acción de “Reanudar el juego”.

{Reanudar el juego}

2. El sistema ejecuta el flujo alternativo del caso de uso **{Pausar el juego}**.
3. El sistema deshabilita la opciones e interfaz del menú de pausa.

FLUJO ALTERNATIVO: -

CASO DE USO: SALIR DEL NIVEL

DESCRIPCIÓN

El jugador puede salir de un nivel empezado y volver al menú principal del juego. Para ello debe seleccionar esta opción desde el menú de pausa de un nivel.

ACTORES

Jugador

PRECONDICIONES

Estar actualmente en el menú de pausa.

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador selecciona la opción de “Salir del nivel”.

{Salir del nivel}

2. El sistema carga el menú principal y vuelve al mismo estado en que se ejecutó.

FLUJO ALTERNATIVO: -

CASO DE USO: VER AYUDA

DESCRIPCIÓN

En cualquier momento de un nivel, el jugador puede seleccionar el apartado de ayuda para que se muestren una pequeña descripción con los conceptos teóricos tratados en ese nivel y pistas de cómo solucionar las diferentes partes de código modificable para alcanzar la salida.

ACTORES

Jugador

PRECONDICIONES: -

FLUJO PRINCIPAL

{Inicio de caso de uso}

1. El jugador selecciona, en cualquier momento del nivel, la acción de “Ver ayuda” disponible en la interfaz.

{Ver ayuda}

2. El sistema ejecuta el caso de uso **{Pausar el juego}**.
3. El sistema muestra por pantalla un bloque de texto redactado específicamente para ayudar y dar pistas en ese nivel.
4. El jugador puede moverse por el bloque de texto hasta que seleccione la opción de salir.
5. El sistema cierra la pantalla de ayuda y se vuelve al mismo punto donde estaba la acción del juego.

FLUJO ALTERNATIVO: -

7. EDITOR DE CÓDIGO

7.1. DESCRIPCIÓN

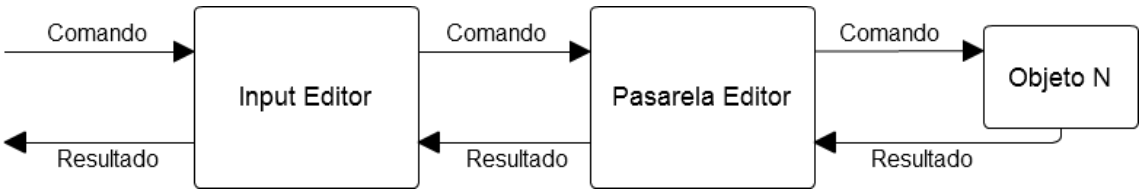
El editor de código que se ha implementado en el videojuego es la principal característica que hace único al control. Permite la interacción entre el usuario y las diferentes partes del juego que se han definido para ser modificadas. Básicamente es una interfaz GUI de Unity[10][11] donde se cargan las partes de código modificables y que cuenta, también, con un InputEditor que permite al usuario introducir un texto que sustituya al actual definido en el código.

Como se ha dicho anteriormente, no ha sido requisito funcional en ningún momento la creación de un complejo editor que permita la compilación de cualquier tipo de comando introducido. Esto solo ya podría constituir un proyecto en sí mismo y no ha sido para nada la voluntad de intentar acercarse a ese alcance.

Se ha diseñado pensando en la accesibilidad de cara al usuario, mostrándole la información de forma permanente, y con un nivel de funcionalidad básico que permitiera modificar texto y validar que la entrada de información sea válida.

7.2. ARQUITECTURA DEL EDITOR

Teniendo en cuenta estas consideraciones a la hora de abordar la implementación del editor, se optó por el siguiente modelo de arquitectura de clases.



Como se puede ver es un sencillo modelo con tres clases involucradas principalmente.

Por una parte, el InputEditor constituye la entrada de información por parte del jugador hacia el videojuego en forma de cadenas de texto. En este punto no hay ningún tipo de validación del comando introducido.

Como pasarela de la información está la clase PasarelEditor que recoge la información transmitida a través del InputEditor y la envía a la clase del objeto al que hace referencia el comando introducido. También sirve para devolver el estado de la ejecución hacia el InputEditor de nuevo y que éste pueda mostrar en consonancia un mensaje de error o hacer efectivo el cambio del texto del código en el editor.

Por último están las diferentes clases a las que les llega la información del comando. Son estas las encargadas de validar que el código esté dentro de los parámetros establecidos. Esta decisión de diseño viene dada porque la validación es propia para cada tipo de clase y bastante diferenciada entre ellas, aunque es cierto que comparten algunas validaciones de sintaxis comunes.

Una vez validado el comando en la propia clase, se efectúa el cambio en el propio código si ha sido favorable y se devuelve el resultado para que la parte de control del InputEditor pueda mostrar un error o cambiar el texto.

7.3. DEFINICIÓN DEL CÓDIGO MEDIANTE XML[17]

Para poder definir y editar cómodamente las partes que el usuario puede ir obteniendo en las diferentes fases, se ha optado por usar el formato XML por su sencillez, eficiencia y fácil obtención de datos mediante Unity. Para cada una de las fases se ha creado un archivo de texto en formato XML con la siguiente estructura:

```
<codigoFase>
    <codigoObjeto>
        <idCodigo> 1 </idCodigo>
        <codigo>

        </codigo>
    </codigoObjeto>
    <codigoObjeto>
        <idCodigo> 2 </idCodigo>
        <codigo>

        </codigo>
    </codigoObjeto>
    <codigoObjeto>
        <idCodigo> 3 </idCodigo>
        <codigo>

        </codigo>
    </codigoObjeto>
</codigoFase>
```

`<codigoFase>`**:** Engloba los códigos de una misma fase por si se quiere tener un solo archivo XML con todos los códigos juntos, aunque se ha optado por separar en diferentes ficheros para que sea más directo acceder al contenido que se quiere modificar.

<codigoObjeto>: Engloba el identificador y el código que pertenecen a un mismo objeto en Unity.

<idCodigo>: Entero que sirve para identificar el código con el objeto de Unity que lo carga en el editor. Solo hace falta que sea único en el archivo pero puede estar repetido entre fases ya que la programación de juego se encarga de discernir cual debe cargar según la fase en la que se encuentre.

<codigo>: El código que se cargará en el editor. Contiene diferentes separadores que identifican los saltos de línea y las partes interactivas según su tipo dentro del juego.

Unity tiene predefinida una estructura de directorios concreta para poder acceder a elementos externos desde el propio script de un elemento del juego. La ruta es: /Assets/Resources. Basta con guardar los ficheros en la ruta especificada y acceder mediante la función

```
Resources.Load("CodigoColeccionable/CodigoFase0"+nivel) as TextAsset;
```

desde código.

Como se explica más adelante, debido a las peculiaridades del parseo del contenido de los campos del fichero (concretamente el campo <codigo>) no debe modificarse los saltos de línea del fichero de texto. Debido al uso de separadores para delimitar las partes de código interactivas así como los saltos de línea dentro del editor del juego, una normalización de los saltos de línea del fichero provocaría problemas a la hora de visualizarlo dentro del juego. Esto es especialmente significativo ya que el editor MonoDevelop pregunta por ello cada vez que el archivo es abierto provocando, como me ocurrió en una ocasión, errores que pueden ser difícilmente localizables.

7.4. SEMÁNTICA DEL XML

La semántica usada en el XML del código de cada nivel es muy simple. A continuación se describen los diferentes separadores y se pone un ejemplo de un código de uno de los niveles para ilustrarlo.

Representa los saltos de línea dentro del texto. Hacen que el documento XML sea legible.

$¡ Inicia el principio de creación de un texto editable. El final viene marcado por el separador $ o por el final de la línea si no se incluye ninguno más

$¿ Igual que el anterior pero con la peculiaridad de que este identifica al texto como una función que puede ser invocada pero no modificada.

Un ejemplo de código usado con esta sintaxis es el siguiente;

```
<codigoFase>
    <codigoObjeto>
        <idCodigo> 1 </idCodigo>
        <codigo>
        #var puerta {
        #    var cerrada = $¡true;
        #    var color = $¡red;
        #}
        </codigo>
    </codigoObjeto>
    <codigoObjeto>
        <idCodigo> 2 </idCodigo>
        <codigo>
        #var plataforma {
        #    var escalaX = $¡1.0;
        #    function $¿resetearEscala();
        #}
        </codigo>
    </codigoObjeto>
    <codigoObjeto>
        <idCodigo> 3 </idCodigo>
        <codigo>
        #var bug {
        #    var vivo = $¡true;
        #}
        </codigo>
    </codigoObjeto>
</codigoFase>
```

7.5. PASOS DE USO PARA CREAR UN CÓDIGO EDITABLE

La metodología empleada para la creación e implementación de un código editable pasa por los siguientes pasos:

1. Crear el código en el XML correspondiente con la sintaxis creada específicamente para el proyecto.

2. Crear los scripts necesarios en el código de Unity que implementen los cambios en el código y las validaciones pertinentes.

3. Crear las nuevas llamadas en el código de la clase PasarelaEditor y enlazarlas con las funciones creadas en los scripts de los objetos afectados.

7.6. MENSAJES DE ERROR

Los mensajes de error son mostrados por pantalla cuando la validación de un comando ha sido negativa. Se introdujeron en la implementación con la intención de facilitar la compresión de lo que estaba ocurriendo en pantalla al jugador. Suelen venir acompañados de pequeñas indicaciones para poder evitarlos notificando el tipo de comando que se espera. Son visibles durante un corto espacio de tiempo y no necesitan de intervención del jugador para desaparecer, intentando informar sin resultar molestos.

8. IMPLEMENTACIÓN

8.1. LISTADO DE CLASES

A continuación se listan todas las clases generadas para la implementación del juego y una breve descripción de cada una de ellas. En el siguiente apartado se analizará con mayor detenimiento el código de las que he considerado más relevantes en la implementación.

CLASE: CAMARACONTROL.JS

DESCRIPCIÓN

Clase que se encarga de mantener la cámara fijada en la posición del personaje actualizando constantemente su posición.

ÁMBITO

Global. Se usa en todas las escenas del juego.

INTERACCIÓN CON OTRAS CLASES O COMPONENTES

Jugador

CLASE: CODIGOITEM.JS

DESCRIPCIÓN

Clase que contiene el identificador de un código para generar un TextoEditable con el contenido apropiado. También se encarga del movimiento del objeto al que está asociada la clase.

ÁMBITO

Global. Se usa en todos los niveles del juego.

REFERENCIAS

Ninguna

CLASE: CONTROLFASES.JS

DESCRIPCIÓN

Clase que gestiona las transiciones entre pantallas del menú principal. Permite mediante el uso de botones de la interfaz, moverse a la pantalla de Instrucciones, Créditos o empezar el juego en cualquier de sus niveles.

ÁMBITO

Menú principal

REFERENCIAS

Cámara – ya que la transición hacia la pantalla de créditos se basa en un movimiento de la cámara.

CLASE: CONTROLJUEGO.JS

DESCRIPCIÓN

Clase principal que se encarga de gestionar la mayoría de eventos como el pausado del juego, el menú de pausa, transiciones entre niveles y, sobre todo, la carga de códigos desde los XML y la instanciación de TextosEditables para incorporarlos al editor.

ÁMBITO

Global. Se usa en todos los niveles del juego.

REFERENCIAS

Jugador, Cámara, PuertaFinal, TextoEditable, Editor

CLASE: CONTROLPAUSA.JS

DESCRIPCIÓN

Clase que ejerce de interfaz entre la lógica de pausa del juego y los eventos activados al clicar en los botones de pausa. También se encarga de gestionar la parte gráfica del pausado.

ÁMBITO

Global. Se usa en todos los niveles del juego.

REFERENCIAS

ControlJuego

CLASE: INPUTEDITOR.JS

DESCRIPCIÓN

Clase que se encarga de gestionar toda la entrada y modificación de comandos hacia las partes de código editables, compuestas por instancias de TextoEditable. También incluye las funciones destinadas a habilitar y deshabilitar el componente de entrada de texto de la interfaz.

ÁMBITO

Global. Se usa en todos los niveles del juego.

REFERENCIAS

PasarelaEditor, ControlJuego, ControlPausa, MensajeError.

CLASE: PASARELAEDITOR.JS

DESCRIPCIÓN

Clase importante que ejerce de pasarela entre los comandos introducidos mediante el InputEditor y la clase final a la que pertenece el código que se quiere modificar. También sirve como enlace para mostrar los mensajes de error que se generan en la clase modificada y el InputEditor.

ÁMBITO

Global. Se usa en todos los niveles del juego.

REFERENCIAS

Multitud – Todos los objetos específicos de un nivel concreto: Puerta, PlataformaInicial, Bug, Bug02, Trampilla, Plataforma, Bug03, PlataformaGiratoria, PlataformaCarga, Deadlock, BugFinal.

CLASE: MENSAJEERROR.JS

DESCRIPCIÓN

Clase sencilla que muestra un texto de error en la interfaz durante un intervalo de tiempo determinado.

ÁMBITO

Global. Se usa en todos los niveles del juego.

REFERENCIAS

Ninguna

CLASE: TEXTOEDITABLE.JS

DESCRIPCIÓN

Clase que sirve para crear el tipo de texto que poder ser modificable mediante el editor del juego. También se encarga de resaltar visualmente el texto seleccionado.

ÁMBITO

Global. Se usa en todos los niveles del juego.

REFERENCIAS

InputEditor, ControlJuego, ControlPausa.

CLASE: JUGADORMOVIMIENTO.JS

DESCRIPCIÓN

Clase importante que se encarga de gestionar el movimiento del personaje y los eventos relacionados con las colisiones e interacciones con el entorno del nivel.

ÁMBITO

Global. Se usa en todos los niveles del juego.

REFERENCIAS

ControlJuego, BugFinal

CLASE: PORTAL.JS

DESCRIPCIÓN

Clase asociada al elemento ubicado al final de los niveles que te permite acceder al siguiente nivel. Representa la salida de cada uno de los niveles excluyendo el final.

ÁMBITO

Global. Se usa en todos los niveles del juego menos en el final (nivel 4).

REFERENCIAS

ControlJuego

CLASE: PUERTA.JS

DESCRIPCIÓN

Clase que gestiona las puertas que se pueden encontrar en los diferentes niveles. En primera instancia se creó como objeto específico del primer código modificable al que se enfrenta el jugador, pero se reaprovechó en otros niveles. Para ello se creó una función directa que abriese la puerta sin tener que recurrir a las acciones del jugador.

ÁMBITO

Global. Se usa en la mayoría de niveles.

REFERENCIAS

Ninguna

CLASE: PUERTAFINAL.JS

DESCRIPCIÓN

Clase asociada a las puertas situadas al final de cada nivel. Lleva el recuento de bugs con vida dentro del nivel para poder gestionar su apertura en caso de que ya no queden más.

ÁMBITO

Global. Se usa en todos los niveles del juego menos en el final (nivel 4).

REFERENCIAS

Ninguna

CLASE: PLATAFORMAINICIAL.JS

DESCRIPCIÓN

Clase simple con la que interactuar en el primer nivel. Se puede modificar su escala y devolverla a la forma inicial mediante una función directa.

ÁMBITO

Nivel 1

REFERENCIAS

Ninguna

CLASE: BUG.JS

DESCRIPCIÓN

Clase del primer bug del juego. Muy simple, solo se encarga de habilitar o deshabilitar el objeto según el comando recibido.

ÁMBITO

Nivel 1

REFERENCIAS

ControlJuego

CLASE: PLATAFORMA.JS

DESCRIPCIÓN

Clase de la plataforma del segundo nivel que permite modificar su rotación y, según la configuración de sus variables, moverse entre las posibilitando el acceso entre las dos habitaciones donde se encuentra.

ÁMBITO

Nivel 2

REFERENCIAS

Ninguna

CLASE: BUG02.JS

DESCRIPCIÓN

Clase que sirve para gestionar el código de los dos bugs existentes en el nivel 2. Por una parte gestiona la modificación de código de uno de ellos y también se encarga de comprobar la condición de muerte por caída del otro bug del nivel.

ÁMBITO

Nivel 2

REFERENCIAS

ControlJuego

CLASE: TRAMPILLA.JS

DESCRIPCIÓN

Clase asociada a la trampilla central de una estancia del nivel 2. Contiene código parecido al usado en el ControlJuego ya que también debe hacer una carga de archivos XML para comprobar si debe abrirse.

ÁMBITO

Nivel 2

REFERENCIAS

Bug

CLASE: BOT.JS

DESCRIPCIÓN

Clase que implementa los pequeños robots que siguen al personaje en el nivel 3. De forma similar a la clase CamaraControl, básicamente se encarga de actualizar el destino de los robots con la posición actual del personaje.

ÁMBITO

Nivel 3

REFERENCIAS

Jugador

CLASE: BUG03.JS

DESCRIPCIÓN

Clase del bug del nivel 3. Se encarga de su eliminación si se cumples las condiciones necesarias especificadas por las funciones que contiene.

ÁMBITO

Nivel 3

REFERENCIAS

ControlJuego

CLASE: PLATAFORMAGIRATORIA.JS

DESCRIPCIÓN

Clase de la plataforma del nivel 3. Su función basa en rotar según el eje que se haya especificado para poder acceder a salas inaccesibles de otra forma. También incluye un método para devolverla a la posición original.

ÁMBITO

Nivel 3

REFERENCIAS

Ninguna

CLASE: PLATAFORMACARGA.JS

DESCRIPCIÓN

Clase de la plataforma que sirve como báscula y gestiona la apertura de una puerta cercana según el número de personajes que estén en contacto con ella. Además incluye la gestión del texto que aparece en la mini pantalla que posee, actualizando cada vez que un objeto entra o sale de su superficie.

ÁMBITO

Nivel 3

REFERENCIAS

Bot, Puerta.

CLASE: DEADLOCK.JS

DESCRIPCIÓN

Clase del último nivel que plantea inicialmente un código con ejecución de dos hilos concurrentes los cuales reservan cada uno de ellos un recurso que necesita el otro, creando una situación de bloqueo mutuo.

ÁMBITO

Nivel 4

REFERENCIAS

Puerta, Puerta2.

CLASE: PUERTA02.JS

DESCRIPCIÓN

Clase vacía que requerida únicamente como segundo recurso a reservar en la situación planteada de exclusión mutua.

ÁMBITO

Nivel 4

REFERENCIAS

Ninguna

CLASE: BUGFINAL.JS

DESCRIPCIÓN

Clase del enemigo final del juego, el cual implementa un sistema de hilos de ejecución asignados por el jugador para poder eliminarlo.

ÁMBITO

Nivel 4

REFERENCIAS

Ninguna

8.2. CLASES RELEVANTES

En este apartado se muestra el código de las clases más relevantes detallando algunas de las funciones más importantes.

CONTROL JUEGO

El control del juego tiene las funciones que implementan la carga de código a través de archivos XML. A continuación se adjunta la parte del código relativa a ésta funcionalidad.

```
/* Función que carga el codigo de los objetos coleccionables directamente des
de el Xml donde está
    especificado, según el nivel actual */
function CargarXmlCodigo()
{
    //Variable para cargar el archivo Xml
    var xmlDocumento : XmlDocument = new XmlDocument();
    //Especificamos el archivo Xml que se quiere cargar
    archivoCodigoXml = Resources.Load("CodigoColeccionable/CodigoFase0"+nivel
) as TextAsset;
    //Cargamos el contenido del fichero Xml
    xmlDocumento.LoadXml(archivoCodigoXml.text);
    //Obtenemos la lista de nodos principales del fichero
    var xmlListaNodos : XmlNodeList = xmlDocumento.GetElementsByTagName("codi
goObjeto");
    for (nodoPadre in xmlListaNodos)
    {
        //Declaración explicita de la conversión para evitar warnings
        var nodoPadreTipado : XmlNode = nodoPadre as XmlNode;
        //Obtenemos los nodos que pertenecen a cada objeto
        var textoLista : XmlNodeList = nodoPadreTipado.ChildNodes;

        //Variables donde guardar temporalmente los datos de los nodos
        var idCodigo : int;
        var textoCodigo : String;

        for (nodo in textoLista)
        {
            //Declaración explicita de la conversión para evitar warnings
            var nodoTipado : XmlNode = nodo as XmlNode;

            //Si el nodo tratado es del tipo id, se convierte a entero y se g
uarda
            if (nodoTipado.Name == "idCodigo")
            {
                idCodigo = parseInt(nodoTipado.InnerText);
            }
            //Sino es el nodo que contiene el texto del código
            else
            {
                textoCodigo = nodoTipado.InnerText;
            }
        }
```

```
        //Almacenamos en la posición del id, el texto relativo a ese identifi
cador del objeto
        codigosFase[idCodigo] = textoCodigo;
    }
}




/* Añade el texto de almacenado con un identificador concreto al editor */
public function PonerTextoCodigo (idCodigo : int )
{
    // Definimos el separador que usamos para indicar los saltos
    var separadorSalto : char[] = ["#"[0]];

    // Conversión explicita del objeto a String para para poder realizar el p
arseo
    // y dividimos la cadena segun el salto de linea especificado
    var textos = (codigosFase[idCodigo] as String).Split(separadorSalto);

    // Definimos el identificador de textos editables
    var separadorEditable : char[] = ["$"[0]];

    for (var value : String in textos)
    {
        // Si la línea de código tiene un identificador editable, creamos el
prefab necesario
        if (value.Contains("$"))
        {
            // Separamos la línea de código
            var codigosEditables = value.Split(separadorEditable);

            var anchuraTexto : float = 0;

            for (var cod : String in codigosEditables)
            {

                if (cod.StartsWith("¡") || cod.StartsWith("¿"))
                {
                    var interactividad : boolean = cod.StartsWith("¡");

                    if (!interactividad) cod = cod.Replace("¿","");
                    else cod = cod.Replace("¡","");

                    var textoCodigoEditable = CrearTextoEditable(idCodigo.ToS
tring(), cod, interactividad);
                    textoCodigoEditable.GetComponent(RectTransform).anchoredP
osition = new Vector2(-1,1);

                    textoCodigoEditable.transform.position += new Vector3 (an
churaTexto, -(numeroFilasEditor * alturaFuente), 0);
                    for (var i = 0; i < cod.length; i++)
                    {
                        editor.text += "  ";
                    }
                }
                else
                {
                    editor.text += cod;
```

```
                }
                anchuraTexto += cod.length * anchuraFuente;
            }
        }
        // Si no, lo añadimos directamente
        else
        {
            editor.text += " " + value;
        }
        // En cualquier caso, aumentamos el numero de filas introducidas para
 poder ubicar correctamente los prefabs creados
        numeroFilasEditor++;
        editor.text += "\n";
    }
    idPrefabActual = 0;
}
```

También incluye, entre otras, la siguiente función destinada a instanciar un objeto de TextoEditable e incorporarlo a los existentes en el editor.

```
/* Instanciación de un nuevo prefab que contenga el código editable de una lí
nea */
function CrearTextoEditable (idClase : String, texto : String, interactivo :
boolean)
{
    // Crear la instancia del nuevo prefab
    var nuevoTexto = Instantiate(textoEditable);


    // Ubicar el elemento correctamente en la jerarquía
    nuevoTexto.transform.SetParent(editor.transform);

    // Ponemos el identificador del prefab para poder trabajar con el posteri
ormente
    nuevoTexto.GetComponent(TextoEditable).idPrefab = idClase + idPrefabActua
l;

    // Añadimos el texto que nos viene dado por el Xml
    nuevoTexto.GetComponent(Text).text = texto;

    nuevoTexto.GetComponent(TextoEditable).interactivo = interactivo;

    ++idPrefabActual;

    return nuevoTexto;
}
```

INPUT EDITOR

El input editor se encarga de mostrar gráficamente las selecciones de texto y ejecutar el primer paso para la modificación según lo visto en la arquitectura del editor. A continuación se detalla el código entero empleado para esta clase:

```
#pragma strict
import UnityEngine.EventSystems;

/* Script que se encarga de la entrada de comandos por el editor */

/* Variables públicas */
public var idPrefab : String = "";
public var textoEditor : Text;

public var pasarelaEditor : PasarelaEditor;
public var controlJuego : ControlJuego;
public var controlPausa : ControlPausa;
public var mensajeError : MensajeError;

/* Variables privadas */
private var editorTexto : InputField;
private var objetoTexto : TextoEditable;
private var temporizadorError : float = 72f;
private var textoErroneo : boolean;
private var sonido : AudioSource;

function Start ()
{
    textoErroneo = false;
    editorTexto = GetComponent(InputField);
    sonido = GetComponent(AudioSource);
}

function Update ()
{
    if (textoErroneo)
    {
        if (temporizadorError <= 0)
        {
            textoEditor.color = Color.black;
            temporizadorError = 72f;
        }
        --temporizadorError;
    }

}

public function PonerTexto(idPrefab : String, texto : String)
{
    editorTexto.interactable = true;
    this.idPrefab = idPrefab;
    editorTexto.text = texto.Trim();

    ResaltarActual();
}
```

```
public function CambiarTextoEnPrefab()
{
    var resultadoEjecucion : String = pasarelaEditor.EnviarCodigo(idPrefab, t
extoEditor.text);

    if (resultadoEjecucion == "ok")
    {
        var prefabs = GameObject.FindGameObjectsWithTag("TextoEditable");
        for (var prefab : GameObject in prefabs)
        {
            if(prefab.GetComponent(TextoEditable).idPrefab == idPrefab)
            {
                prefab.GetComponent(Text).text = textoEditor.text;
                prefab.GetComponent(Text).color = Color.blue;
                break;
            }
        }
        Pausar(false);
        HabilitarInput(false);
        PlaySonido();
    }
    else
    {
        textoEditor.color = Color.red;
        textoErroneo = true;
        mensajeError.ActivarMensaje(resultadoEjecucion);
    }
}

//Sobrecarga de la función sin parámetro de entrada para los eventos
public function ResaltarActual()
{
    ResaltarActual(idPrefab);
}

public function ResaltarActual(idPrefab : String)
{
    var prefabs = GameObject.FindGameObjectsWithTag("TextoEditable");
    for (var prefab : GameObject in prefabs)
    {
        var oPrefab = prefab.GetComponent(TextoEditable);
        if(oPrefab.idPrefab == idPrefab)
        {
            oPrefab.ResaltarTextoSeleccionado();
        }
        else oPrefab.PonerColorNormal();
    }
}

public function HabilitarInput(activar : boolean)
{
    editorTexto.interactable = activar;
    if (!activar)
    {
        editorTexto.text = "";

        //Quitamos el focus del textbox para que se pueda volver a mover el j
ugador de forma normal
```

```
        //y no lo interprete como texto a introducir en el editor.
        EventSystem.current.SetSelectedGameObject(null);
    }
}

public function EjecutarNoInteractivo(idPrefab : String)
{
    var resultadoEjecucion : String = pasarelaEditor.EnviarCodigo(idPrefab, t
extoEditor.text);

    if (resultadoEjecucion != "ok")
    {
        textoEditor.color = Color.red;
        textoErroneo = true;
        mensajeError.ActivarMensaje(resultadoEjecucion);
    }
    else PlaySonido();
}


public function Pausar(pausar : boolean)
{
    controlJuego.PausarJuego(pausar);
    controlPausa.Pausar(pausar);
}

private function PlaySonido()
{
    sonido.Play();
}
```

PASARELA EDITOR

Esta parte de código es una muestra de la pasarela editor actuando para los comandos correspondientes al segundo nivel.

```
public function EnviarCodigoNivel2 (idPrefab : String, texto : String)
{
    var ejecucion : String;

    switch(idPrefab)
    {

        case "10":
            ejecucion = bug02.SetVida(texto);
            break;

        case "11":
            ejecucion = bug02.SetVariableVida(texto);
            break;

        case "30":
            ejecucion = trampilla.AbrirXML(texto);
            break;
```

```
        case "40":
            ejecucion = plataforma.SetCase01(texto);
            break;

        case "41":
            ejecucion = plataforma.SetCase02(texto);
            break;

        default:
            ejecucion = "Prefab no encontrado";
    }

    return ejecucion;
}
```

THREADING[12][13]

El contenido de la clase usada en el último bug detalla el uso de threading[14] para ejecutar esta parte del juego.

```
/* Script del último bug del juego que usa threads para poder derrotarlo */

/* Variables públicas */
public var thread : Thread;
public var thread2 : Thread;
public var thread3 : Thread;

// Objetos referenciados con los que interactua
public var barrera : GameObject;
public var barreraAbierta : GameObject;
public var barreraCerrada : GameObject;
public var barraSalud : GameObject;
public var saludSlider : Slider;
public var celebracion : GameObject;

/* Variables privadas */
// Variables para que los threads se puedan comunicar con el thread principal
private var salud : float;
private var barreraActiva : boolean;

private var funcion1 : String;
private var funcion2 : String;
private var funcion3 : String;


function Start ()
{
    salud = 99;
    barreraActiva = true;

    funcion1 = "Idle";
    funcion2 = "Idle";
    funcion3 = "Idle";
}

function Update ()
```

```
{
    // Actualiza la salud del enemigo y la muestra si es necesario
    saludSlider.value = salud;
    barrera.SetActive(barreraActiva);
    barreraCerrada.SetActive(barreraActiva);
    barreraAbierta.SetActive(!barreraActiva);

    // Al morir el bug, se deja caer por el vacío y se pone en marcha la cele
bración
    if (salud <= 0 )
    {
        transform.position.z += 0.03;
        GetComponent(Rigidbody).isKinematic = false;
        celebracion.SetActive(true);
    }

}

public function Atacar()
{
    while (salud >= 0)
    {
        if (!barreraActiva) salud--;
        Thread.Sleep(200);
    }
}

public function Idle()
{
    Thread.Sleep(99999999);
}


public function Desactivar()
{
    while (true)
    {
        barreraActiva = false;
        Thread.Sleep(200);
    }
}

public function CambiarFunciones(comando : String, numeroFuncion : int)
{
    switch(numeroFuncion)
    {
        case 1:
            funcion1 = comando;
            break;

        case 2:
            funcion2 = comando;
            break;

        case 3:
            funcion3 = comando;
            break;

        default:
```

```
            break;
    }

    return "ok";
}


public function StartThreads()
{
    //Asociamos la función a cada thread segun los valores que tengan actualm
ente en el editor
    if (funcion1 == "Atacar") thread = Thread(Atacar);
    else if (funcion1 == "Quitar") thread = Thread(Desactivar);
    else thread = Thread(Idle);

    if (funcion2 == "Atacar") thread2 = Thread(Atacar);
    else if (funcion2 == "Quitar") thread2 = Thread(Desactivar);
    else thread2 = Thread(Idle);

    if (funcion3 == "Atacar") thread3 = Thread(Atacar);
    else if (funcion3 == "Quitar") thread3 = Thread(Desactivar);
    else thread3 = Thread(Idle);

    // Se inician los threads
    thread.Start();
    thread2.Start();
    thread3.Start();

    return "ok";
}

/* Función para parar los threads usando el metodo Abort */
public function StopThreads()
{
    if (thread.IsAlive)
    {
        thread.Sleep(200);
        thread.Abort();
    }
    if (thread2.IsAlive)
    {
        thread2.Sleep(200);
        thread2.Abort();

    }
    if (thread3.IsAlive)
    {
        thread3.Sleep(200);
        thread3.Abort();

    }

    return "ok";
}

/* Función que nos asegura que si se quita la aplicación, los threads dejen d
e funcionar */
public function OnApplicationQuit()
{
```

```
    if (thread.IsAlive)
    {
        thread.Sleep(200);
        thread.Abort();
    }
    if (thread2.IsAlive)
    {
        thread2.Sleep(200);
        thread2.Abort();
    }
    if (thread3.IsAlive)
    {
        thread3.Sleep(200);
        thread3.Abort();
    }
}

public function MostrarVida()
{
    barraSalud.SetActive(true);
}
```

En este aspecto cabe destacar la complicación a la hora de intentar realizar diseños con concurrencia en Unity porque, aunque permite su implementación mediante la clase System.Threading de .NET[14], no posibilita interactuar con ningún elemento de la API de Unity.

La API de Unity no está preparada para dar soporte a threads y por tanto, no se puede usar ninguna de sus funciones ni modificar directamente los componentes u objetos del juego. Se hicieron pruebas intentando mostrar de forma más gráfica el flujo de los threads que intervienen en esta parte del juego, representándolos mediante los objetos Bot del tercer nivel. La poca flexibilidad de Unity en este aspecto ha hecho que tuviese que desistir ya que no era viable implementar tantos cambios que afectasen continuamente a otros objetos[15].

Para poder realizar este código, los threads han tenido que interactuar con el resto de elementos del juego mediante variables globales. El hilo principal de Unity se encargaba de gestionar los cambios en base al valor de esas variables.

9. PARTE GRÁFICA DEL JUEGO

9.1. DESCRIPCIÓN

El videojuego se basa en modelos 3D simples en escenarios generados con salas predefinidas. La vista es cercana a una cámara cenital desde donde se puede ver los movimientos del personaje en tercera persona de forma cómoda. En todo momento la cámara está centrada en el personaje y le sigue en sus movimientos.

Para el arte del juego se ha optado por una solución simple pero con un resultado convincente, el modelado de escenarios y personajes mediante *voxels*.

Los voxels son, en esencia, cubos unitarios que permiten construir fácilmente un modelo de forma análoga a como se haría mediante el coloreado de pixels que se ha usado históricamente para crear sprites en dos dimensiones, pero añadiendo una dimensión de profundidad. De esta forma se consigue el mismo efecto “pixelado” pero en modelaje 3D.

Su uso se ha aplicado al modelado del personaje principal, los enemigos, objetos coleccionables y las diferentes salas y partes que componen el escenario de cada fase. Básicamente, ha sido la técnica de arte principal a usar en la creación del videojuego, a excepción de unos pocos recursos que ya ofrece Unity, como el uso de partículas o fuentes de texto.

MagicaVoxel[8] es la herramienta escogida para la creación de dichos modelos. Se trata de un editor de imagen basado en voxels de uso intuitivo, fácil exportación y uso gratuito. Con él se han creado los diferentes modelados que posteriormente se han exportado a formato obj, uno de los múltiples formatos con integración directa que acepta Unity.

Cada vez que se ha creado un objeto para el videojuego, se ha guardado en el formato propio del programa (.vox) para poder tratarlo posteriormente por si hacía falta, pero también se ha creado el archivo .obj seleccionando de entre las múltiples exportaciones que permite el programa de edición.

Dichos archivos obj han sido importados a Unity como parte de los Assets que conforman el juego, pudiendo ser tratados como un objeto más de Unity y añadiéndoles los componentes que han sido necesarios para su correcto comportamiento.

Los archivos obj contienen toda la información de la geometría del modelo, principalmente: posición de cada vértice, el vector normal, las caras definidas por los polígonos creados mediante los vértices y las coordenadas de las texturas empleadas para ellos. La información

relativa a la textura de las caras, lo que coloquialmente podríamos definir como la paleta usada para pintar cada uno de los voxels, se guarda en otro archivo externo que también debe importarse a Unity como un Asset del tipo Material. De esta forma se puede aplicar una única paleta o material a los diferentes modelos importados ya que corresponde a la usada en el programa de edición MagicaVoxel. Solo sería necesario exportar otra paleta a Unity si se quisiese cambiar los colores de los modelados de forma sencilla. De esta forma tendríamos, por ejemplo, varios tipos de salas con distintos colores pero compartiendo la misma geometría.

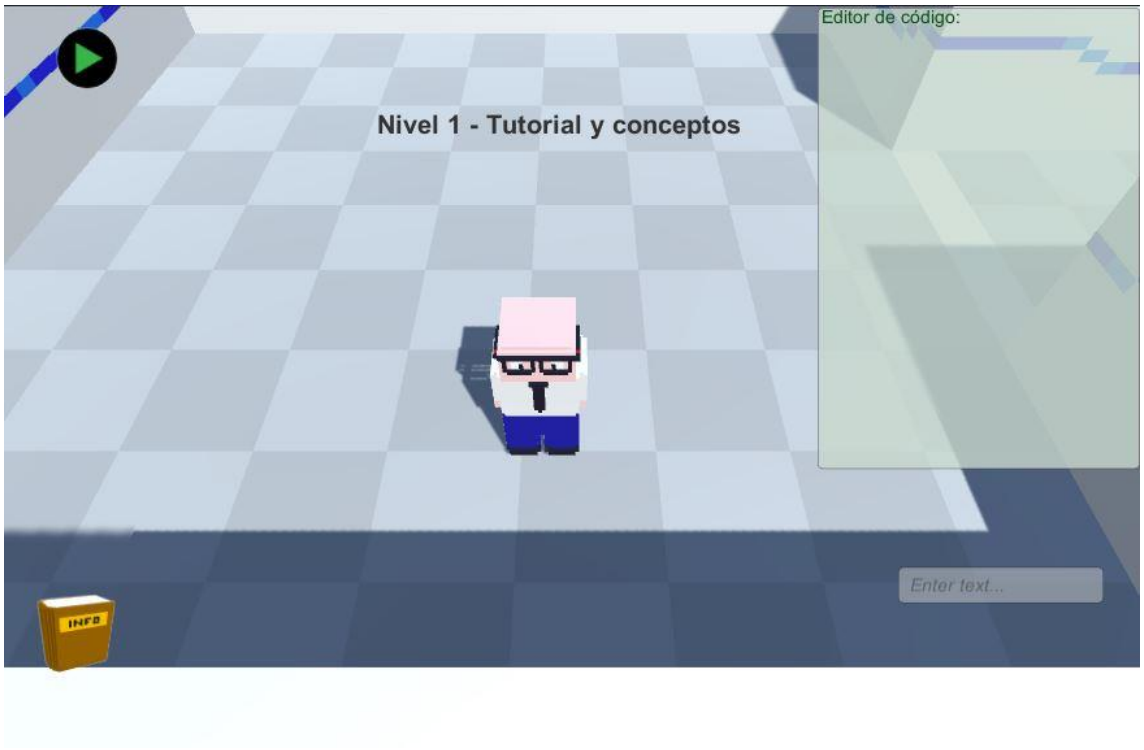
9.2. STORYBOARDS

En este apartado se muestran capturas del resultado final para ilustrar la implementación y la parte gráfica explicada.

El menú principal del juego basado en una escena del inicio añadiéndole el modelado del título y la interfaz con botones.

La pantalla de créditos es similar a la del menú principal ya que se ha aplicado una transición en la cámara que la desplaza lateralmente. Al acabar se muestra los créditos y se da la opción de volver a la pantalla principal del menú.

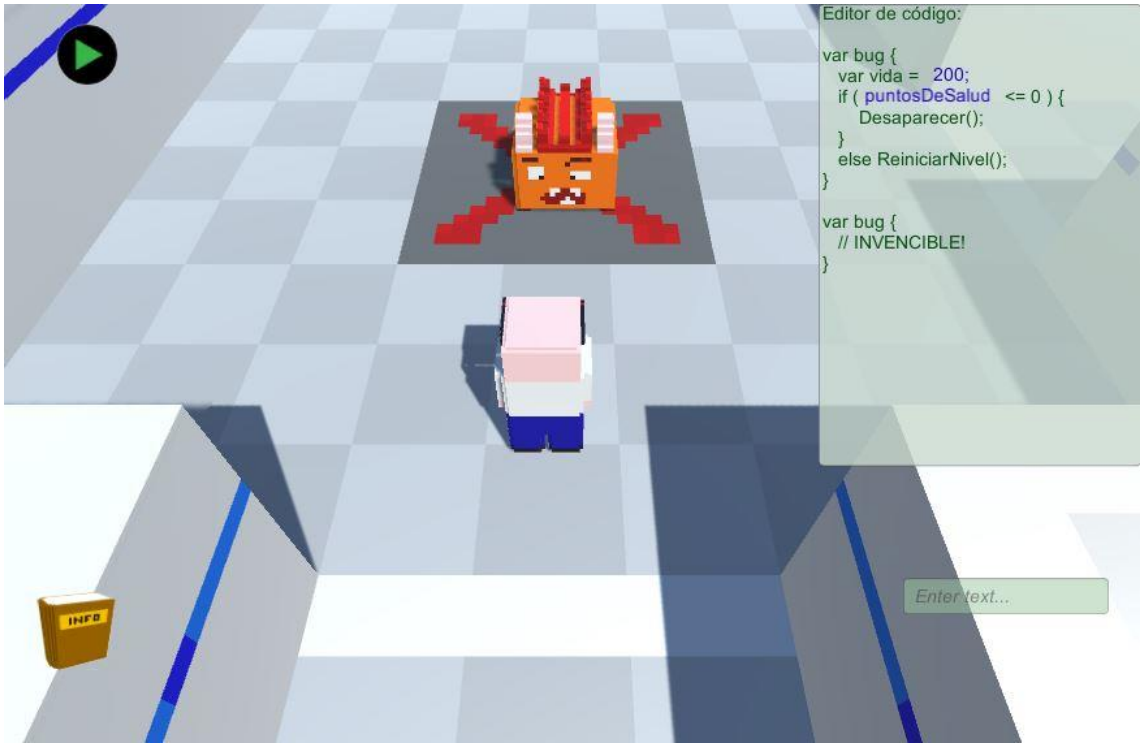
La pantalla de instrucciones es muy básica mostrando el mensaje introductorio para situar al jugador.



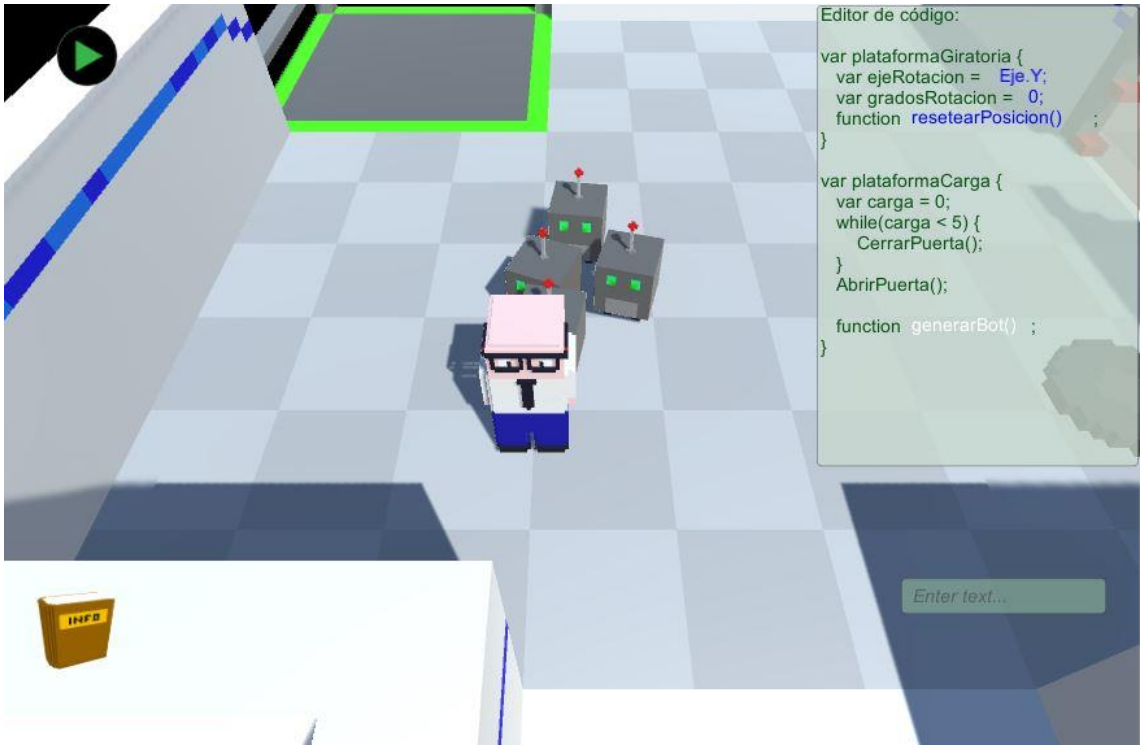
Una muestra del principio del juego, con el editor de código vacío y el título del nivel todavía en pantalla. Se desvanece a los pocos segundos de comenzar.

La opción de ayuda que incorporan los cuatro niveles para introducir levemente el tema y proporcionar pistas y soluciones si el jugador se encalla o desconoce la resolución. En el anexo

de este documento se encuentran la información de todas las situaciones planteadas en el juego.



Detalle de enemigo del segundo nivel.



Robots persiguiendo al personaje en el tercer nivel.

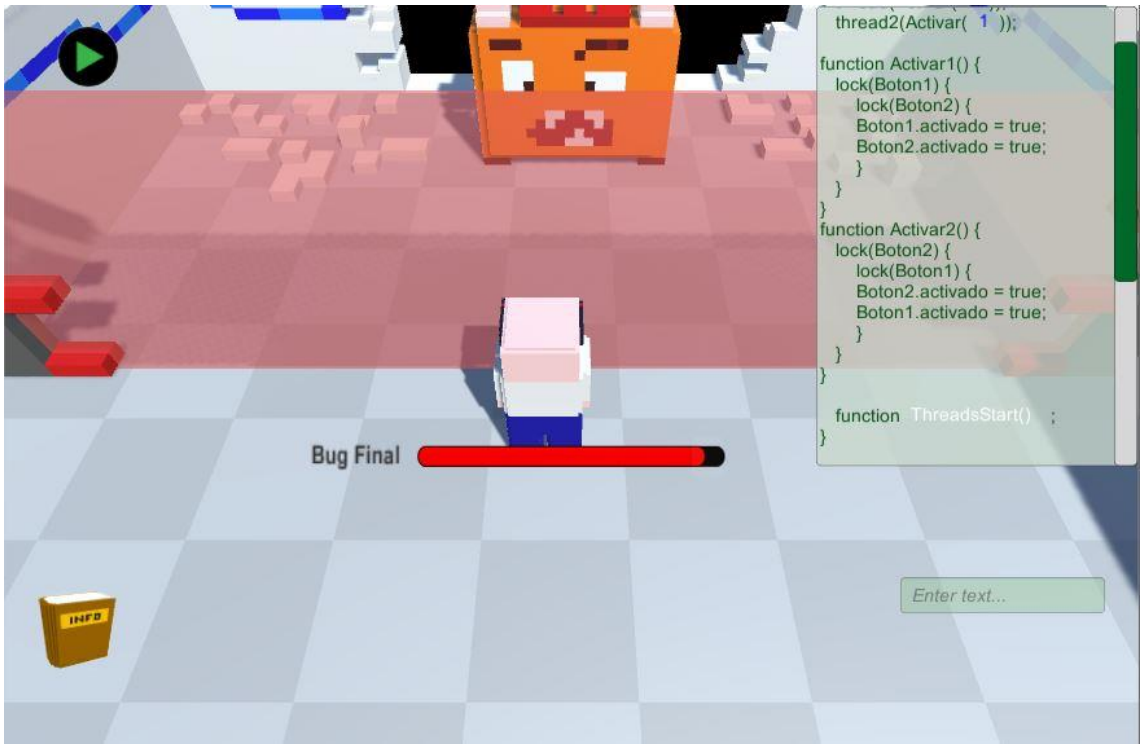


Imagen del enfrentamiento final con el último bug. Se ha de vencer mediante la asignación de funciones a hilos de ejecución.

10. PRUEBAS Y GENERACIÓN DE CÓDIGO

10.1. PRUEBAS

Debido a las características del desarrollo, con ciclos de implementación continuos, no se ha establecido un plan detallado de pruebas. En cada funcionalidad o mejora que se implementaba, se hacían las pruebas necesarias para comprobar su correcto funcionamiento.

Una vez implementadas un paquete de funcionalidades, se pasaba a hacer una prueba de integridad con el resto del nivel. Como los niveles son independientes entre sí, ya que están guardados en diferentes escenas de Unity, con estas pruebas son suficientes para garantizar que no se provocan problemas con el resto de clases y componentes del juego.

En la etapa final del desarrollo, sí que se han hecho muchas más pruebas generales basadas en comprobar la coherencia y la integridad de todos los elementos del juego.

Por desgracia, por falta de tiempo y previsión de contacto con gente que tenga el perfil objetivo del videojuego, no se han podido hacer las pruebas generales con usuarios. Igualmente, me he dado cuenta de que estaban mal planificadas ya que al estar al final de toda la planificación, no quedaba mucho margen para mejoras aunque se hubiesen podido realizar.

10.2. DESPLIEGUE WEB

La forma de distribución del juego elegida ha sido una aplicación web. Como el propio Unity ya da la posibilidad de generar el contenido en este formato, se optó por usar su servicio en la nube con integración continua [16]. De esta forma se tiene un seguimiento de las generaciones de código que se van publicando de forma automática en un su plataforma web.

A la hora de implementar se han tenido que realizar ajustes en la posición de los componentes debido a la menor resolución nativa con la que se suele trabajar en el entorno web. Eso no ha impedido que el resultado sea muy similar al ofrecido en un formato de aplicación de escritorio convencional.

A pesar de haber distribuido el juego como un elemento web como se tenía planeado, este apartado del proyecto ha resultado ser menor del esperado. La planificación inicial contaba con algún tipo de soporte web alojado en un servicio de host que no ha podido realizarse. El resultado pues, aun tratándose de un aplicativo web, no se ha podido modificar para adaptarlo a un entorno más profesional de plataforma de videojuegos web.

10.3. PROBLEMAS CON LAS VERSIONES DE UNITY

A lo largo del proyecto han surgido dos problemas totalmente diferentes con la versión de Unity usada.

El primero de ellos surgió en una fase temprana del desarrollo cuando se intentó usar la funcionalidad de integración continuada en la nube explicada anteriormente. En su última versión estable, Unity ha cambiado la tecnología usada para la portabilidad a plataforma web. Ha pasado de usar la propia que tenía integrada históricamente por tecnología WebGL. Esto ha provocado que todavía no tenga integrado el despliegue automático en la nube. Para poder aprovechar las ventajas del Cloud Unity Build se optó por cambiar a una versión anterior que no provocase errores a la hora de compilar el código remotamente.

La otra problemática, cuyo efecto era más grave, vino dada en la última etapa de desarrollo. En las pruebas rutinarias del juego se experimentaba unas bajadas irregulares pero continuadas en la tasa de imágenes, lo cual provocaba unos molestos “tirones” a la hora de mover al personaje. En un principio creía que podría estar ocasionado por la carga gráfica del juego ya que, aunque los modelos usados son muy básicos, contienen muchos más triángulos de los que deberían debido a la herramienta con la que han sido usados.

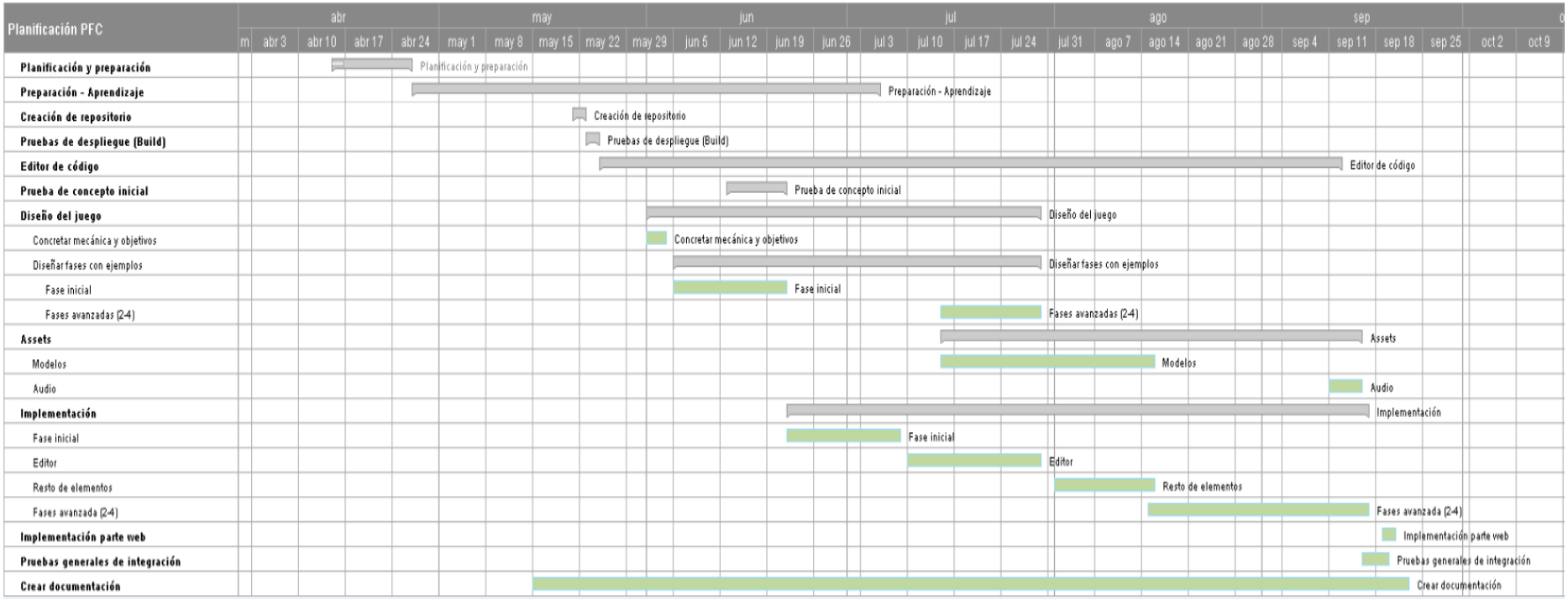
Tras buscar investigar cual era un número razonable de triángulos para una escena de un juego, constaté que los modelos creados estaban igualmente muy por debajo del umbral de la carga poligonal de un modelo corriente.

Después de usar la herramienta de inspección de rendimiento de Unity, detecté que efectivamente la carga de cpu relativa al renderizado de la imagen era menor del 10%, un porcentaje muy aceptable. En cambio, se originaban picos irregulares asociados a la gestión de las físicas, los cuales cuadraban perfectamente con la problemática detectada. Tras buscar información sobre este tipo de problema, encontré que viene dado por un bug interno de Unity que poseen unas pocas versiones lanzadas. Nuevamente se cambió por una versión mayor, aunque inferior a la última para poder seguir usando el Cloud Unity Build, y se solucionó sin mayor altercado.

11. PLANIFICACIÓN FINAL

11.1. PLANIFICACIÓN TEMPORAL

En el siguiente diagrama se muestra la planificación temporal final del proyecto con los tiempos reales de desarrollo. En general ha habido un retraso generalizado a causa del aprendizaje Unity, el cual ha sido más laborioso de lo esperado. Una vez llegado al final de la planificación se ve las tareas de pruebas generales e integración con web han sufrido grandes recortes. Los requisitos funcionales han podido ser llevados a cabo con éxito dentro del margen establecido. La duración estimada del trabajo corresponde con unas 644 horas ya que se estima una media de 4 horas trabajas al día (aunque el fin de semana se hacían más que compensan las de algunos días laborales y unos días que estuve fuera en el mes de agosto).



11.2. PLANIFICACIÓN ECONÓMICA

Teniendo en cuenta que no ha sido necesaria la adquisición de ningún tipo de hardware, que el software que se ha utilizado (principalmente Unity y MagicaVoxel) es gratuito y que los efectos de sonido son libres, los costes económicos del proyecto son esencialmente derivados de los recursos humanos. En este caso, existe un perfil de trabajador principal que es el programador, así como un perfil secundario, el diseñador, aunque en la práctica han sido la misma persona. Con tal de realizar una valoración económica hemos asignado un precio por hora trabajada a cada uno de estos perfiles:

- Programador: 25€/h
- Diseñador: 30€/h

Tarea	Perfil	Horas	Coste
Planificación y preparación	Programador	32h	800€
Preparación - Aprendizaje	Programador	148h	3700€
Creación de repositorio	Programador	4h	100€
Pruebas de despliegue (Build)	Programador	8h	200€
Editor de código	Programador	76h	1900€
Prueba de concepto inicial	Programador	28h	700€
Diseño del juego	Programador	120h	3000€
Assets	Diseñador	150h	4500€
Implementación	Programador	212h	5300€
Implementación parte web	Programador	8h	800€
Pruebas generales de integración	Programador	8h	200€
TOTAL		**644h**	**21.200€**

12. CONCLUSIONES

Al finalizar este proyecto y repasar los objetivos iniciales puedo afirmar que la mayor parte de ellos se ha cumplido satisfactoriamente. De hecho, el resultado se asemeja en gran medida a la idea concebida en un principio. El desarrollo del videojuego ha generado un producto acabado y con cierta entidad propia.

En primer lugar, he logrado plantear un diseño que incorpora la posibilidad de modificación de código por parte del usuario, aportando un enfoque jugable distinto. Su implementación se ha llevado a cabo no sin complicaciones y limitaciones impuestas por el tiempo y conocimiento del motor de desarrollo. A pesar de ello, el resultado cumple con las expectativas propuestas.

En segundo lugar, se ha logrado captar la intención de mostrar los efectos de un código de programación de forma visual mediante las características propias de un videojuego. Por otro lado, la generación del código distribuible ha sido funcional como videojuego web pero con un alcance menor del planeado. Las opciones de exponer el resultado como un elemento web atrayente han sido supeditadas por motivos de prioridad marcados por la limitación temporal.

En todo caso, personalmente, ha sido muy enriquecedor aprender en un ámbito que me era desconocido y que me despertaba mucho interés. Además, puedo concluir que estoy satisfecho con el resultado obtenido y sorprendido gratamente con algunos aspectos como por ejemplo, la parte gráfica desarrollada y el aspecto visual obtenido.

Concluyo este proyecto con muy satisfecho y con ganas renovadas de enfocar mi vida laboral al mundo de los videojuegos.

13. MEJORAS DEL PROYECTO

En mi opinión el proyecto podría ampliarse claramente en dos facetas diferenciadas:

- Ampliar el número de niveles y situaciones presentadas en el videojuego con nuevo "temario" añadido. Especialmente interesante me parece la posibilidad de expandir su universo con nuevos mundos diferenciados en los que la temática recurrente sea otra. Por ejemplo, se podría adaptar a las comunicaciones en red, poniendo a prueba el reconocimiento de puertos, redes y protocolos que darían pie a nuevas situaciones para resolver por parte del jugador pero con una mecánica no muy alejada de la presentada en este proyecto.

- Por otra parte, la otra ampliación clave sería el aspecto menos desarrollado al final de este trabajo: editar e incorporar funcionalidades a una plataforma web que incorpore el juego creado. Ampliando este campo se podría llegar a tener una aplicación web con diferentes videojuegos educativos, cada uno de ellos desarrollado de forma análoga a éste pero con peculiaridades específicas según el tipo de conocimiento que se quiera tratar.

14. REFERENCIAS

[1] Página oficial del motor Unity http://unity3d.com

[2] Definición de Gamificación en Wikipedia https://es.wikipedia.org/wiki/Ludificaci%C3%B3n

[3] Gamificación - https://hipertextual.com/archivo/2015/01/que-es-gamificacion/

[4] Scratch - Página oficial https://scratch.mit.edu/

[5] CodeCombat - Página oficial https://codecombat.com/

[6] Hack'n Slash - Página oficial http://www.hacknslashthegame.com/

[7] CodeHunt - Página oficial https://www.codehunt.com/

[8] Herramienta de diseño gráfico basado en voxels - MagicaVoxel - http://voxelart.blogspot.com.es/2015/02/magicavoxel-programa-editar-voxel.html

[9] Metodología ágil Scrum - https://es.wikipedia.org/wiki/Scrum_(desarrollo_de_software)

[10] Manual de Unity http://docs.unity3d.com/es/current/Manual/index.html

[11] API de Unity https://docs.unity3d.com/es/current/ScriptReference/index.html

[12] Programación concurrente - Entrada Wikipedia - https://es.wikipedia.org/wiki/Computaci%C3%B3n_concurrente

[13] Ejemplo de bloqueo mutuo

http://www.jasoft.org/Blog/post/191;Que-es-un-deadlock-o-interbloqueo.aspx

[14] Multi-threading en .NET -

http://www.yoda.arachsys.com/csharp/threads/

[15] Uso de threads en Unity - http://answers.unity3d.com/questions/357033/unity3d-and-c-coroutines-vs-threading.html

[16] Tutorial de despliegue de repositorios para Unity - https://unity3d.com/es/learn/tutorials/topics/cloud-build/creating-your-first-source-control-repository

[17] Ficheros XML en Unity

http://forum.unity3d.com/threads/xml-reading-a-xml-file-in-unity-how-to-do-it.44441/

[18] Ejemplos de juegos con temática similar –

http://www.bloglenovo.es/la-programacion-es-un-juego-de-ninos-o-eso-intentan-estas-catorce-propuestas/

[19] Lista de juegos creados con Unity - https://en.wikipedia.org/wiki/List_of_Unity_games

[20] Tutoriales 2D y 3D de Unity - https://unity3d.com/es/learn/tutorials/projects/roll-ball-tutorial

[21] Toda la música ha sido descargada gratuitamente de:

http://freemusicarchive.org/

[22] Todos los efectos de sonido han sido descargados gratuitamente de:

https://www.freesound.org

ANEXO: DISEÑO DE LOS NIVELES

Para cada uno de los niveles se han diseñado diferentes escenarios usando las salas y elementos previamente almacenados como si de piezas físicas encajables se trataran.
Se ha limitado a cuatro niveles diferentes en los cuales se trabajarán elementos concretos de cualquier lenguaje de programación. Aun así, se ha optado por representar el código en lenguaje JavaScript (el mismo que verdaderamente lo implementa en Unity) con algunas licencias pensadas en dar mayor legibilidad y ocupar un menor espacio, haciéndolo más accesible al jugador.

La estructuración en niveles permite introducir progresivamente nuevos elementos o conceptos con los que tiene que tratar el jugador además de mantener, si se desea, los conceptos que ya se han ido viendo anteriormente. De esta forma se crea una curva de dificultad que atiende a razones puramente teóricas y no a un aumento de la exigencia en las mecánicas del videojuego, como ocurre en la mayoría de ellos.

A continuación se exponen los diseños finales de los cuatro niveles que componen el juego con detalle en las modificaciones de código necesarias y los conceptos teóricos trabajados.

NIVEL 1: TUTORIAL Y CONCEPTOS BÁSICOS

El primer nivel del juego sirve como tutorial y toma de contactos con las mecánicas jugables. Los conceptos que se presentan son los cambios en los valores de asignaciones de variables ya establecidas. De esta forma se obliga al jugador a afrontar simples puzles o situaciones que requieren un simple conocimiento de los tipos de variables comunes. De esta forma mediante el diseño, nos aseguramos que capta la idea del juego de cara a niveles más complejos.

En concreto las situaciones a resolver son las siguientes;

```
var puerta {
      var cerrada = true;
       var color = red;
}
```

El primer código modificable está justo delante de la puerta a la que referencia.
Se debe cambiar el valor booleano de la variable cerrada para avanzar.
A modo de curiosidad y para que el jugador pueda investigar, se puede cambiar el color del láser que impide el paso por los parámetros definidos para la clase Color de Unity.

```
var plataforma {
        var escalaX = 1.0;
        function resetearEscala();
}
```

En un vacío insalvable se encuentra la plataforma que nos permite pasar modificando su tamaño. También incluye la primera referencia a una función la cual puede ser activada simplemente clicando sobre ella sin necesidad de editar nada. En este caso devuelve la plataforma a su tamaño original.

```
var bug {
        var vivo = true;
}
```

El primer bug del juego se muestra enjaulado y enfrente de la puerta final del nivel que impide el paso hacia la salida si todavía quedan bugs en el escenario.
Simplemente hay que cambiar el valor de la variable.

NIVEL 2: INCORPORA LA CREACIÓN DE VARIABLES Y SENTENCIAS CONDICIONALES

En la segunda fase se introducen las sentencias condicionales. Este tipo de sentencias sirven para redirigir el flujo del código según unas condiciones que se han especificado previamente. De esta forma se consigue que se ejecuten las partes de código deseadas en función de los valores que le llegan como entrada.

En concreto las situaciones a resolver son las siguientes;

```
var bug {
        var vida = 200;
        if (puntosDeSalud <= 0 ) {
            Desaparecer();
        }
        else ReiniciarNivel();
}
```

El primer bug del nivel plantea una situación muy simple con el tipo de sentencia condicional if-else. Si no se presta atención al orden de los cambios que se realizan, el flujo del código hace que se reinicie el nivel.

Para solucionarlo debemos fijarnos que la vida debe sustituirse por cero o un valor negativo y después cambiar el tipo de variable que se compara en la condición por la especificada en la línea de arriba. Si se hiciese con el orden a la inversa, se ejecutaría la otra parte del código ya que el campo vida tiene más valor que cero.

```
var trampilla {
      try {
        var xml = new XmlDocument();
        xml.Load(“CodigoFase02”);
      }
      catch (Exception e) {
        Abrir();
      }
   }
```

El segundo bug es único en el juego ya que no se elimina mediante código, sino que debe hacerse caer abriendo la trampilla en la que está situado. Para ejecutar este código se introduce el concepto de captura de excepciones.

La captura de excepciones puede ser considerada un tipo de elemento condicional de programación ya que, debidamente usadas, pueden ser beneficiosas para el flujo del código, aunque su uso se suela limitar a desenmascarar errores de ejecución.

En este caso se intenta abrir uno de los archivos XML con el código de la fase pero lo que se pretende es que falle para poder pasar por el código de la excepción. Para ello simplemente debemos inventarnos un nombre de fichero ya que no lo encontrará y saltará la excepción.

```
var plataforma {
      var posicion = "vertical";
      switch(posicion){
      case  horizontal:
          PonerHorizontal();
          break;
      case   vertical:
          PonerVertical();
          break;
      default:
          Mover();
      }
   }
```

El último código presenta el tipo de condicional restante, la sentencia de tipo “switch”. Es el equivalente a concatenar una serie de if-else que comprueban el valor de la misma variable y en función del resultado obtenido ejecutan una parte de código u otra.

Para poder avanzar hay que poner el primer caso como “vertical”. De esta forma se ejecuta el primer código ya que coincide con la variable declarada. Luego se han de modificar los dos restantes por los valores que se deseen para que ninguno coincida y el flujo pase por el default permitiendo mover la plataforma.

NIVEL 3: INCORPORA SENTENCIAS ITERATIVAS

En la tercera fase se introducen las sentencias iterativas while y for, junto con un puzle de reorganización de código. Las sentencias iterativas son altamente frecuentes y sirven para ejecutar una porción de código múltiples veces.

```
var plataformaGiratoria {
      var ejeRotacion = Eje.X;
      var gradosRotacion = 0;
      function resetearPosicion();
   }
```

El primer código hace uso simplemente de asignaciones de variables para poder mover una plataforma según los grados especificados y el eje en que se debe mover.

```
var bug {
      var salud = 10;
      function codigo01() {
          salud++;
      }
      function codigo02() {
          for(i=0; i<10; i++) {
              salud--;
          }
      }
      function codigo03() {
          if(salud <= 0) {
              Desaparecer();
          }
      }
   }

var ponerCodigoBug {
    bug.codigo03();
```

```
        bug.codigo01();
        bug.codigo03();
      }
```

El segundo bug hace referencia a una secuencia de código. Para destruirlo simplemente hay que ordenar el código correctamente y hacer que la comprobación del condicional se cumpla.

```
    var plataformaCarga {
          var carga = 0;
          while(carga < 5) {
              CerrarPuerta();
          }
          AbrirPuerta();

          function generarBot();
      }
```

El último código del nivel pone en práctica la el uso de la sentencia while de forma gráfica con la generación de unos pequeños robots que sirven para cumplir la condición de salida y poder abrir la puerta.

NIVEL 4: CONCURRENCIA

El último de los niveles se basa enteramente en la concurrencia y paralelismo de ejecución de código. La concurrencia es la simultaneidad en la ejecución de múltiples tareas o hilos de ejecución creados por un único programa. La programación concurrente está relacionada con la programación paralela, pero enfatiza más la interacción entre tareas. Trata sobre la correcta secuencia de interacciones o comunicaciones entre los procesos y el acceso coordinado de recursos que se comparten por todos los procesos. De esta forma se consigue un uso más eficiente de los recursos físicos disponibles aunque los riesgos y problemáticas de la programación concurrente pueden ser complicados de detectar.

```
    var Bots {
          thread(Activar(1));
          thread2(Activar(2));

       function Activar1() {
          lock(Boton1) {
              lock(Boton2) {
              Boton1.activado = true;
              Boton2.activado = true;
              }
```

```
            }
        }
        function Activar2() {
            lock(Boton2) {
                lock(Boton1) {
                Boton2.activado = true;
                Boton1.activado = true;
                }
            }
        }

            function ThreadsStart();
        }
```

El primer obstáculo del nivel presenta un ejemplo claro de bloque mutuo o deadlock. El bloque mutuo es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente, que compiten por recursos del sistema o bien se comunican entre ellos. A diferencia de otros problemas de concurrencia de procesos, no existe una solución general para los interbloqueos.

El ejemplo es claro, el primer hilo de ejecución accede a un recurso y lo bloquea mientras el segundo hace lo propio con el otro. De esta forma se consigue un cuelgue en el sistema. Para poder solucionarlo hay que serializar el flujo de los hilos haciendo que los dos pasen por las mismas funciones uno detrás del otro.

```
    var bug {
            var thread = Thread1(Idle);
            var thread2 = Thread(Idle);
            var thread3 = Thread(Idle);
            function Atacar() {
                if(!barreraActiva)
                    saludBug--;
            }
            function Quitar() {
                barreraActiva = false;
            }
            function Idle() {
                Thread.Sleep(999999);
            }

            function ThreadsStart();
            function ThreadsStop();
        }
```

El último código del juego presenta una asignación de tres hilos de ejecución que no presentan problemas de bloqueo entre ellos. El jugador puede asignarlos libremente a las funciones

representadas para poder desactivar la barrera y acabar con el enemigo. Primero se puede hacer una asignación basada en “Atacar, Quitar, Atacar” para luego parar los hilos y asignarles la función de “Atacar” a cada uno de ellos. Así se puede comprobar como el rendimiento aumenta eliminando antes al enemigo.